

LA-UR-18-25458

Approved for public release; distribution is unlimited.

Title: Intel x86 Assembly: Malware Analysis Day 2

Author(s): Pearce, Lauren

Intended for: Presentation for a 2 week malware analysis course

Issued: 2018-06-21

Disclaimer:

Los Alamos National Laboratory, an affirmative action/equal opportunity employer, is operated by the Los Alamos National Security, LLC for the National Nuclear Security Administration of the U.S. Department of Energy under contract DE-AC52-06NA25396. By approving this article, the publisher recognizes that the U.S. Government retains nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or to allow others to do so, for U.S. Government purposes. Los Alamos National Laboratory requests that the publisher identify this article as work performed under the auspices of the U.S. Department of Energy. Los Alamos National Laboratory strongly supports academic freedom and a researcher's right to publish; as an institution, however, the Laboratory does not endorse the viewpoint of a publication or guarantee its technical correctness.

Intel x86 Assembly

Malware Analysis Day 2

lauren.p@lanl.gov

License

- Some slides and images from today's lectures come from the course “Introduction to Intel x86 Assembly, Architecture, Applications, and Alliteration” by Xeno Kovah.
- I *highly* recommend this course – it's open and completely free, complete with slides, video lectures, labs, and a chat room.
<http://www.opensecuritytraining.info/IntroX86.html>
- The course was licensed under a Creative Commons “Share Alike” license, meaning this slide deck is licensed under the same.

All materials is licensed under a Creative Commons “Share Alike” license.

- <http://creativecommons.org/licenses/by-sa/3.0/>

You are free:



to Share — to copy, distribute and transmit the work



to Remix — to adapt the work

Under the following conditions:



Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).



Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

Assembly Code

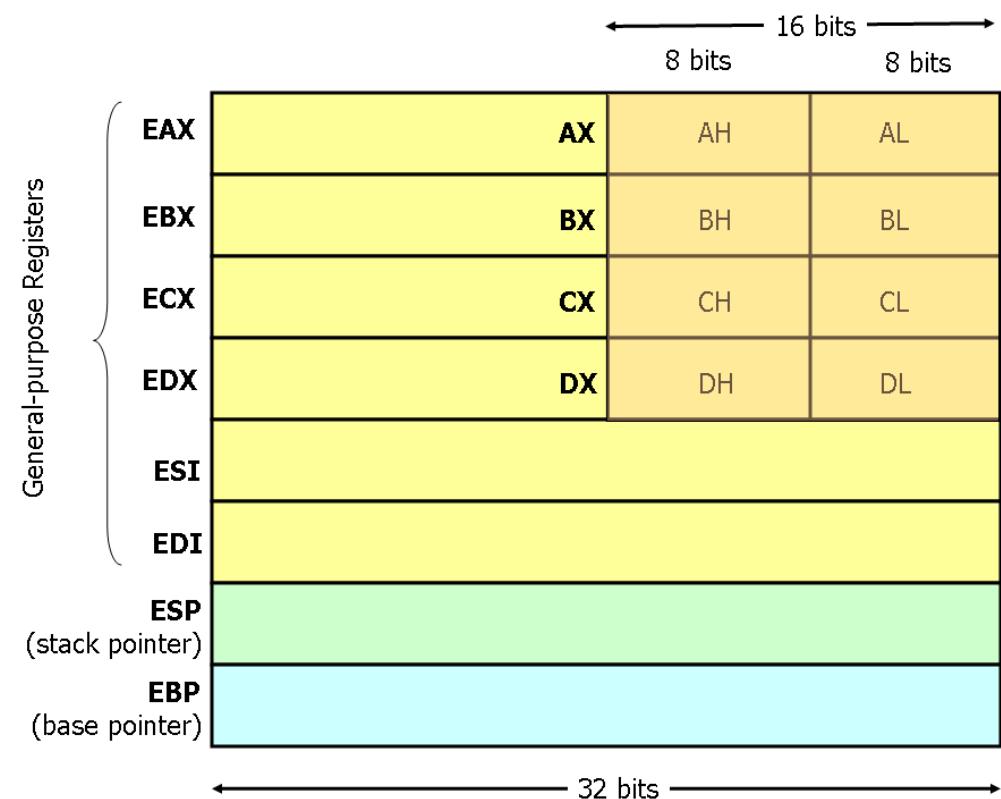
- What is it?
- Why do we need to know how to understand it?
- What other jobs require you to know assembly?
- Why am I teaching x86 instead of x64?

X86 Instruction Format

- An instruction is made of a single mnemonic plus zero or more operands.
- The mnemonic is often shorthand for the instruction, for example:
 - mov is the mnemonic for move
 - shl is the mnemonic for shift left
 - jnz is the mnemonic for jump not zero
- Operands identify the data that the instruction uses. There are 3 types of operands in x86:
 - Immediate: An actual fixed value
 - Register: A reference to a small amount of named storage
 - Memory: A memory address containing the data to be operated on

Registers

- Registers are processor based storage
 - Allow for temporary storage and quick access of small amounts of data
 - Why not just use memory?



Von Neumann Architecture

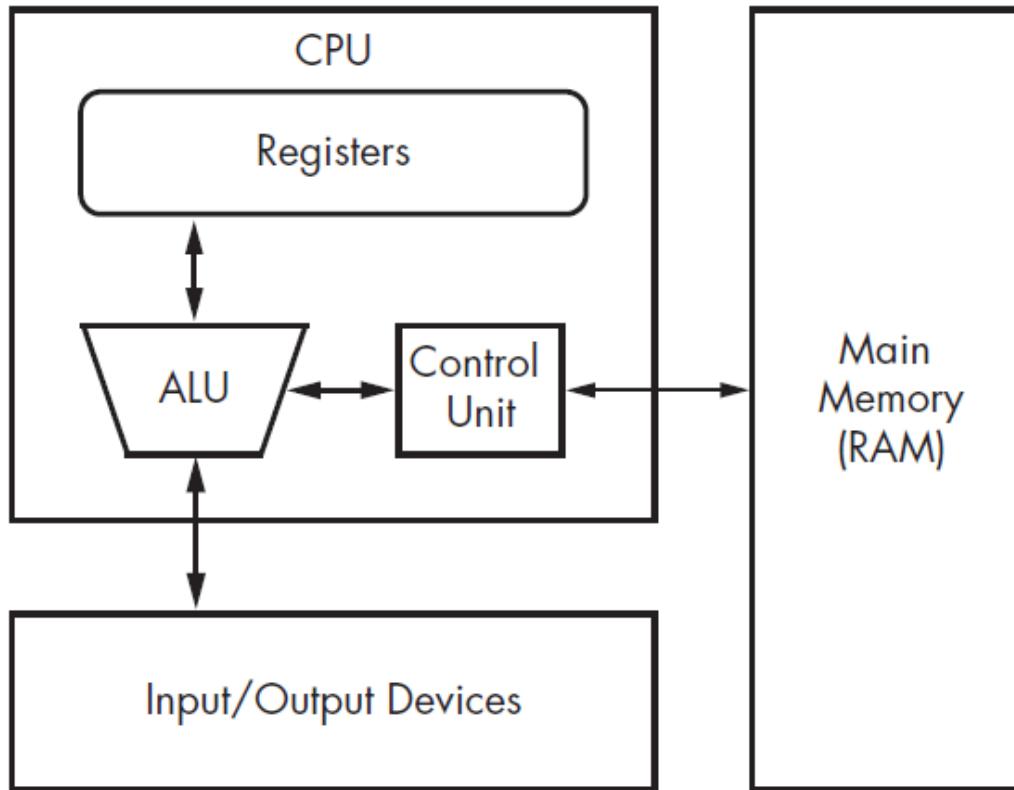


Figure 4-2: Von Neumann architecture

A Few Special Registers

- EIP - EIP holds the memory address of the instruction to be executed next. EIP's **sole purpose in life** is to tell the processor what instruction to execute next.
 - Why is EIP a target for attackers?
- EFLAGS Register
 - Essentially a 32 bit register of single bit flags. Only 2 flags that we care about at the moment:
 - Zero Sign Flag (ZF) – Set if the result of some instruction is zero, cleared otherwise
 - Sign Flag (SF) – Zero indicates positive, one indicates negative.

NOP

- No Operation
- Used to pad bytes
- We use it to patch out instructions that are in our way
- Offense uses it for some basic exploits - google “NOP sled” if you’re curious.

The Stack

- The Stack is LIFO (Last In First Out) data structure. Data is "pushed" onto the top of the stack, then "popped" off the top.
- The stack grows DOWN towards LOWER memory
- ESP is the **Stack Pointer** and always points to the top of the stack.
 - The top of the stack is the _____ address being used by the stack
- The stack tracks what functions were called before the running function, the local variables, and is used to pass function parameters.
- You cannot understand assembly without understanding the stack.

PUSH

- Places an operand on the top of the stack.
- We're adding something to the stack – what else has to change?
- We're adding something to the top of the stack, the stack grows down, so we must decrement the pointer that points to the top of the stack - ESP. So, push actually does two things:
 1. Decrement esp by 4
 2. Place operand into the 32 bit space at address [ESP]

Registers Before

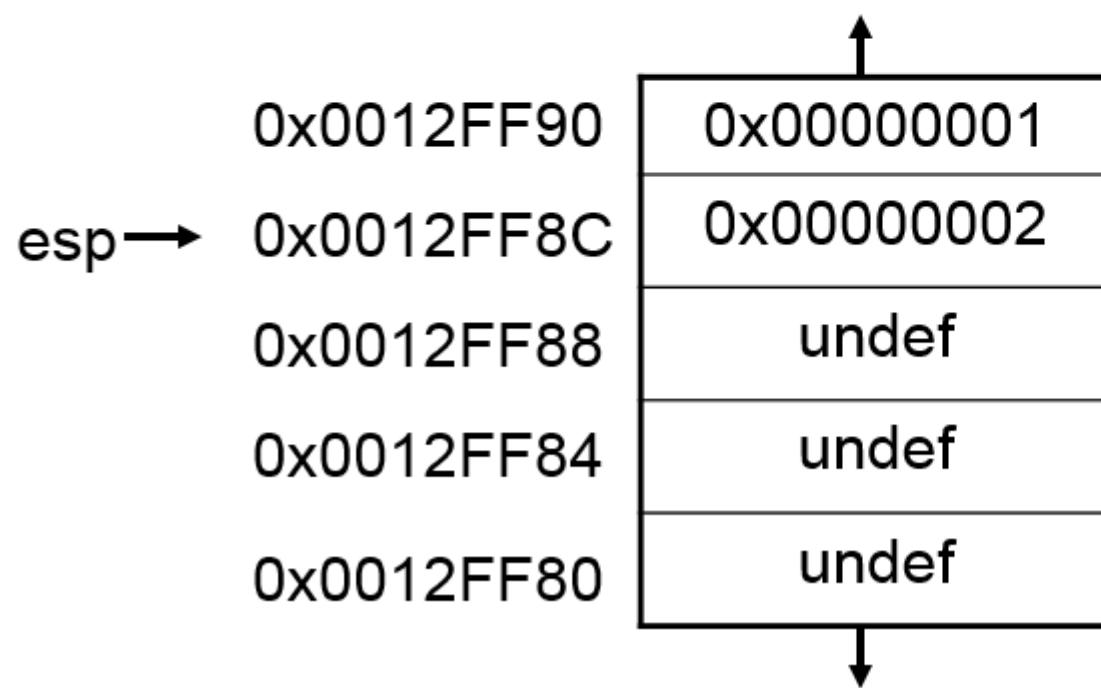
eax	0x00000003
esp	0x0012FF8C

push eax

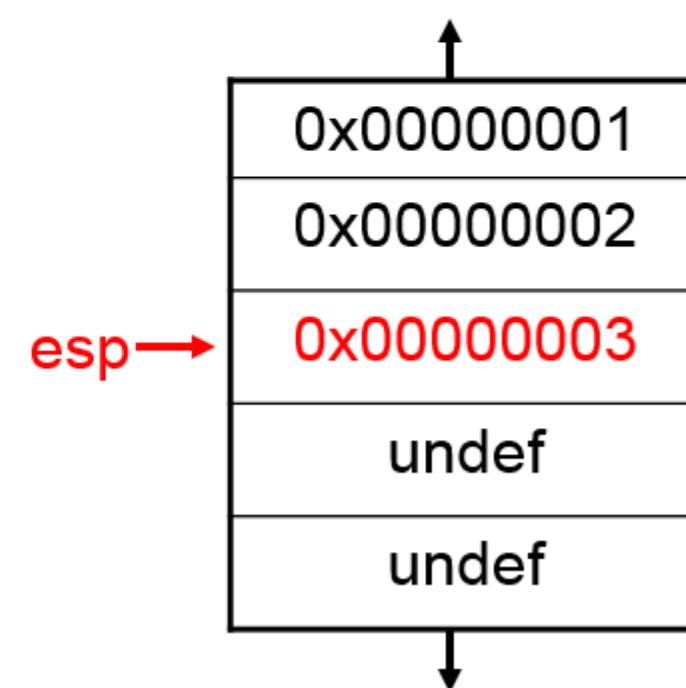
Registers After

eax	0x00000003
esp	0x0012FF88

Stack Before



Stack After



POP

- The pop instruction is the opposite of the push instruction.
- pop removes a 32 bit data element from the top of the stack and increments ESP.
- Understanding Check: $\text{ESP} = 0x0012FF84$ and I execute a pop instruction. Now what does ESP equal?

Registers Before

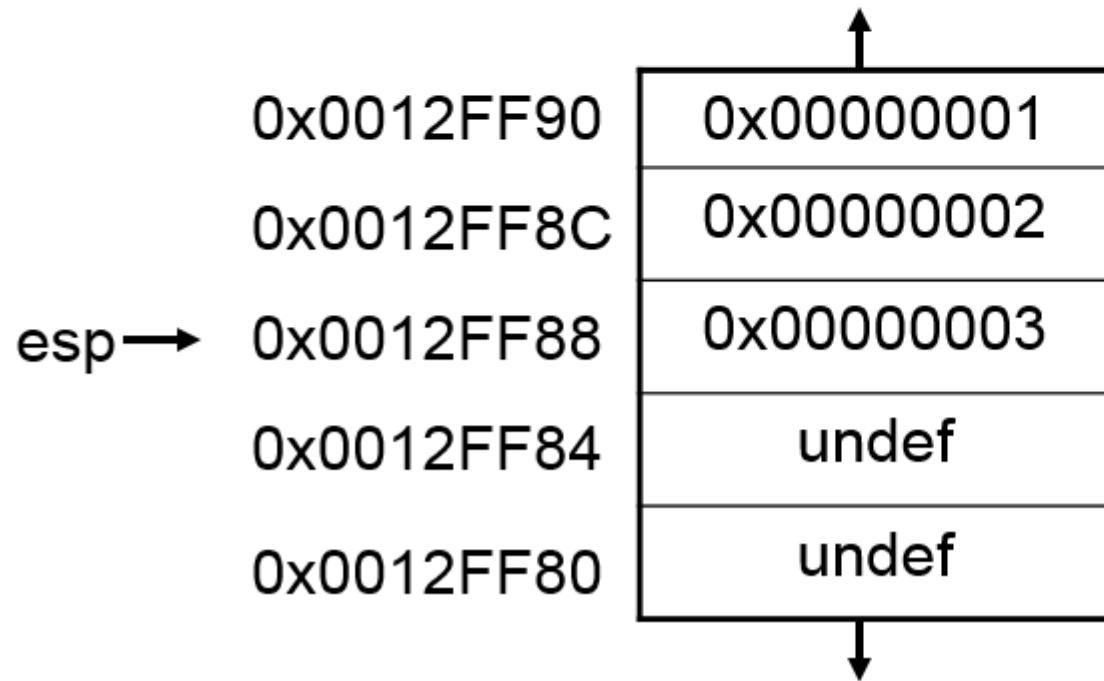
eax	0xFFFFFFFF
esp	0x0012FF88

pop eax

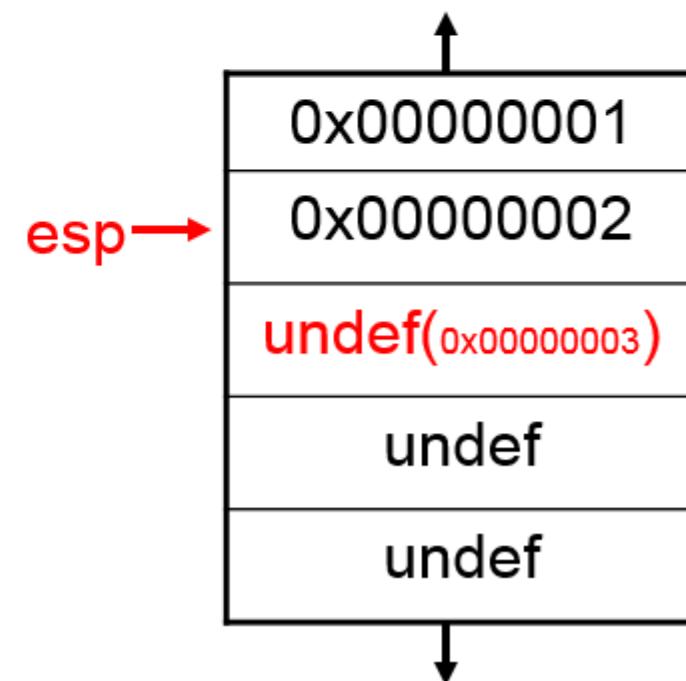
Registers After

eax	0x00000003
esp	0x0012FF8C

Stack Before



Stack After



Calling Conventions

- Refers to how code calls a function
- Compiler dependent and configurable – many different conventions exist.
- We'll talk about cdecl, stdcall, and fastcall

cdecl

- “C declaration” – most common calling convention
- Function parameters are pushed onto the stack **right to left**
- EAX holds the return result
- **Calling** function is responsible for cleaning up the stack

stdcall

- Used by Microsoft's C++ code
- Function parameters pushed onto the stack **right to left**
- eax holds the return result
- **Called** function is responsible for cleaning up the stack

fastcall

- The programmer can specify in his code that fastcall should be used for specific functions.
- The first two DWORD or smaller arguments from left to right are passed in the ECX and EDX registers. All other arguments are passed on the stack from right to left.
- **Called function** cleans the stack

CALL

- CALL's job is to transfer control to a different function, in a way that control can later be resumed where it left off
- First it pushes the address of the next instruction onto the stack
 - For use by RET for when the procedure is done
- Then it changes eip to the address given in the instruction
- Destination address can be specified in multiple ways
 - Absolute address
 - Relative address (relative to the end of the instruction)

RET

- Two forms
 - Pop the top of the stack into eip (remember pop increments stack pointer)
 - In this form, the instruction is just written as “ret”
 - Typically used by cdecl functions
 - Pop the top of the stack into eip and add a constant number of bytes to esp
 - In this form, the instruction is written as “ret 0x8”, or “ret 0x20”, etc
 - Typically used by stdcall functions

MOV

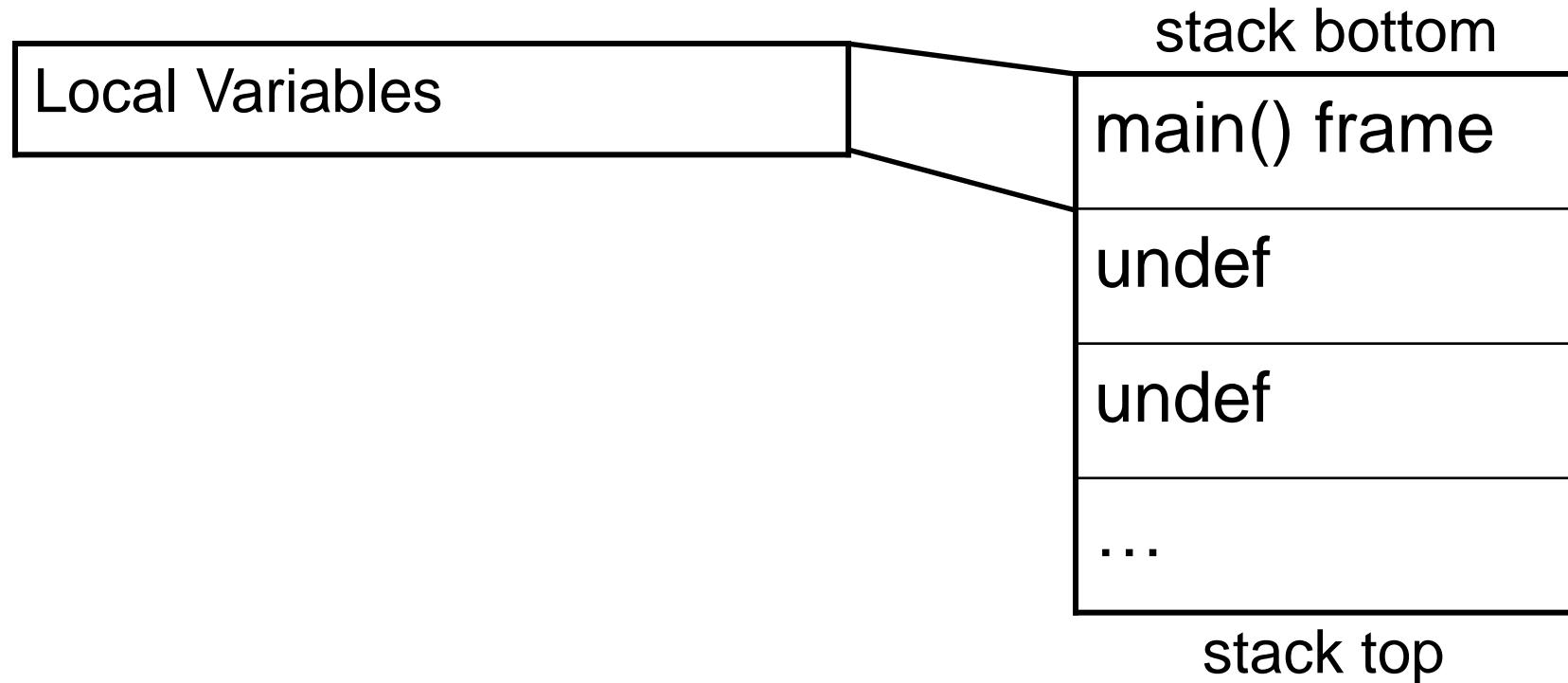
- Can move:
 - register to register
 - memory to register, register to memory
 - immediate to register, immediate to memory
- Never memory to memory!

The Stack!

- Now that we have a few instructions to work with, let's go back to talking about the stack.
- Remember – within the stack, each function has its own “stack frame” that is, as far as the function’s concerned, its very own stack.
- EBP and ESP are the pointers that are responsible for framing the stack – EBP points to the **base** of a function’s stack frame, while ESP points to the top of the stack.

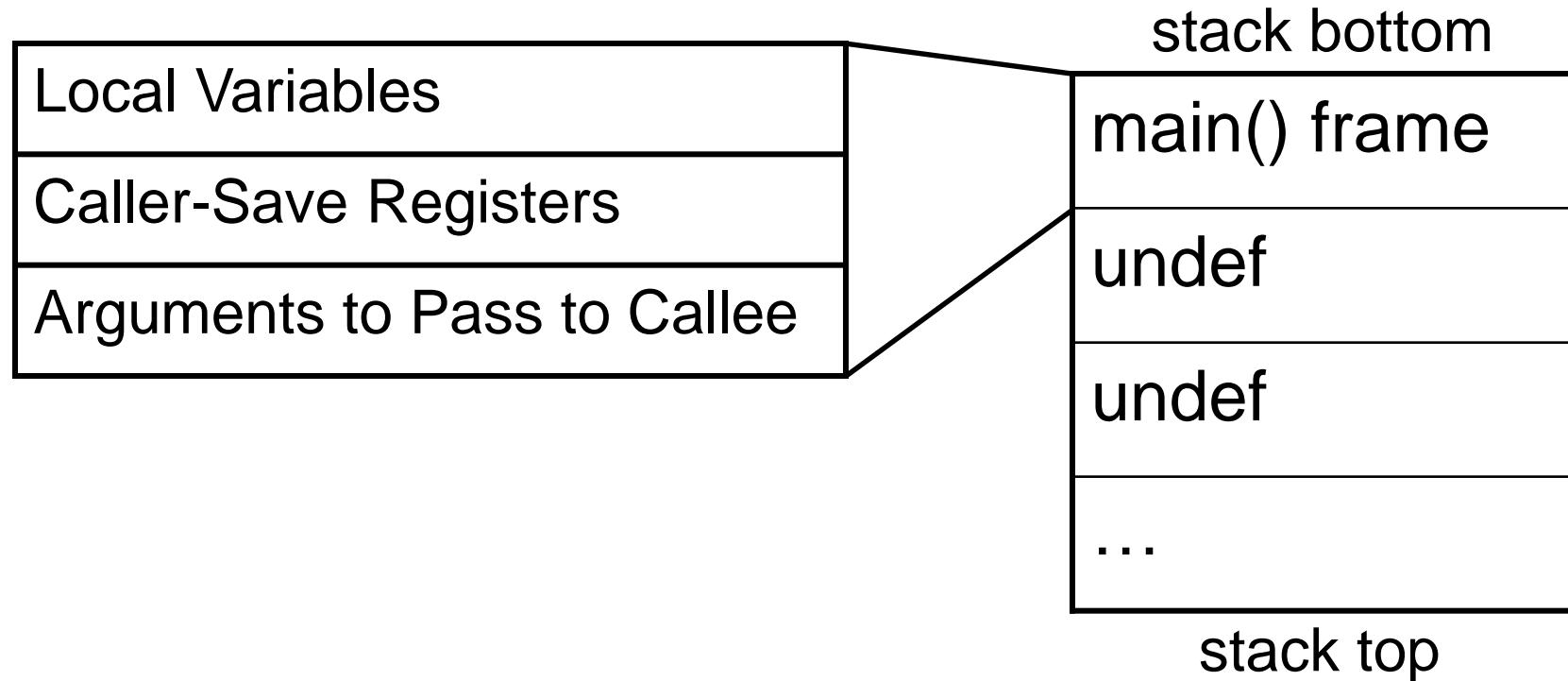
General Stack Frame Operation

We are going to pretend that main() is the very first function being executed in a program. This is what its stack looks like to start with (assuming it has any local variables).



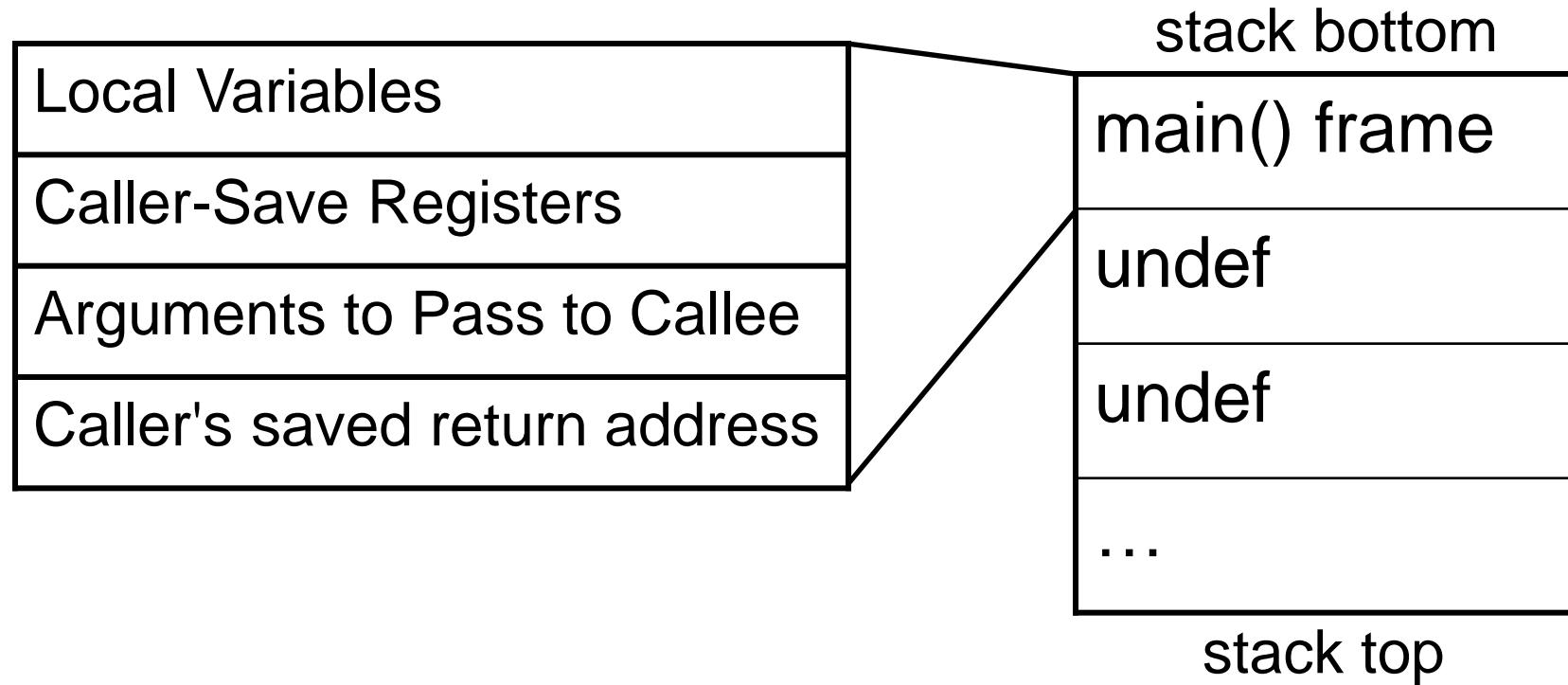
General Stack Frame Operation 2

When `main()` decides to call a subroutine, `main()` becomes “the caller”. We will assume `main()` has some registers it would like to remain the same, so it will save them. We will also assume that the callee function takes some input arguments.



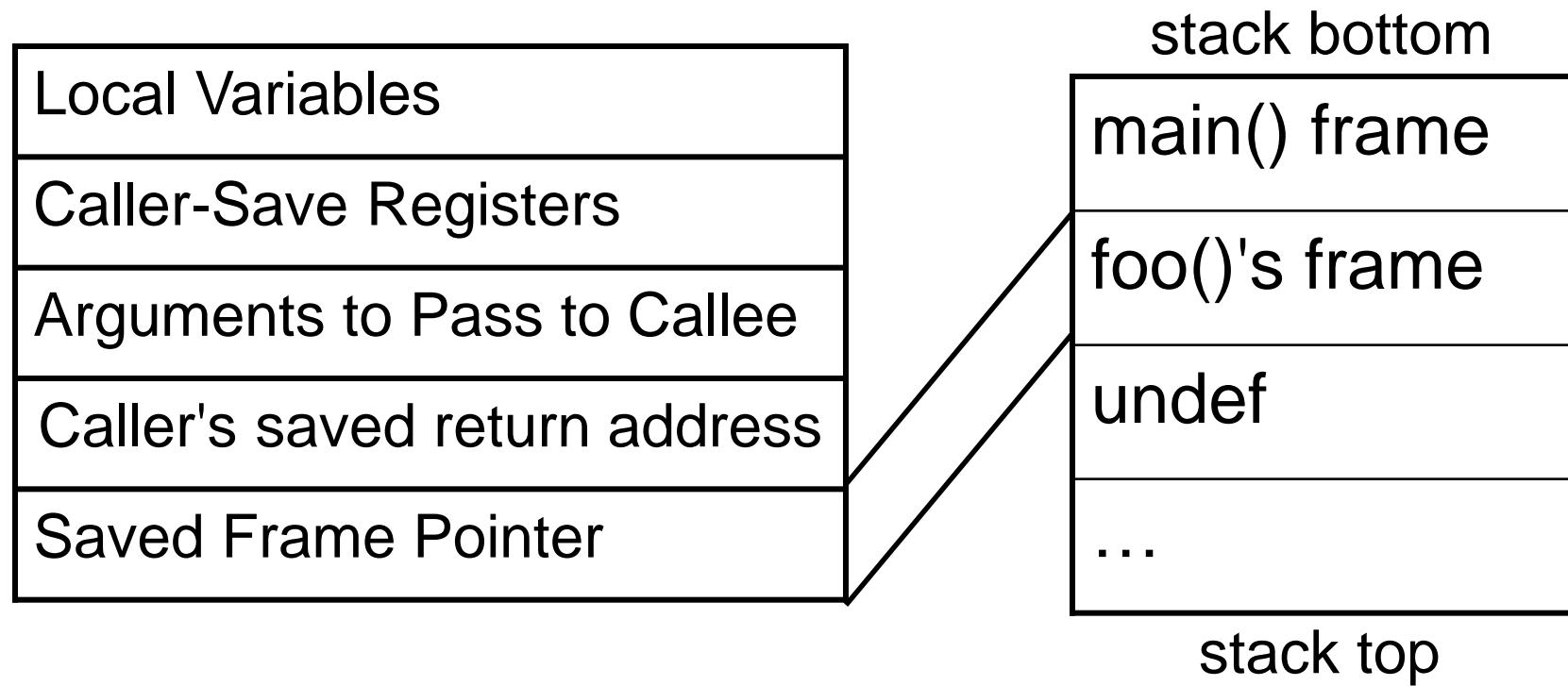
General Stack Frame Operation 3

When `main()` actually issues the `CALL` instruction, the return address gets saved onto the stack, and because the next instruction after the call will be the beginning of the called function, we consider the frame to have changed to the callee.



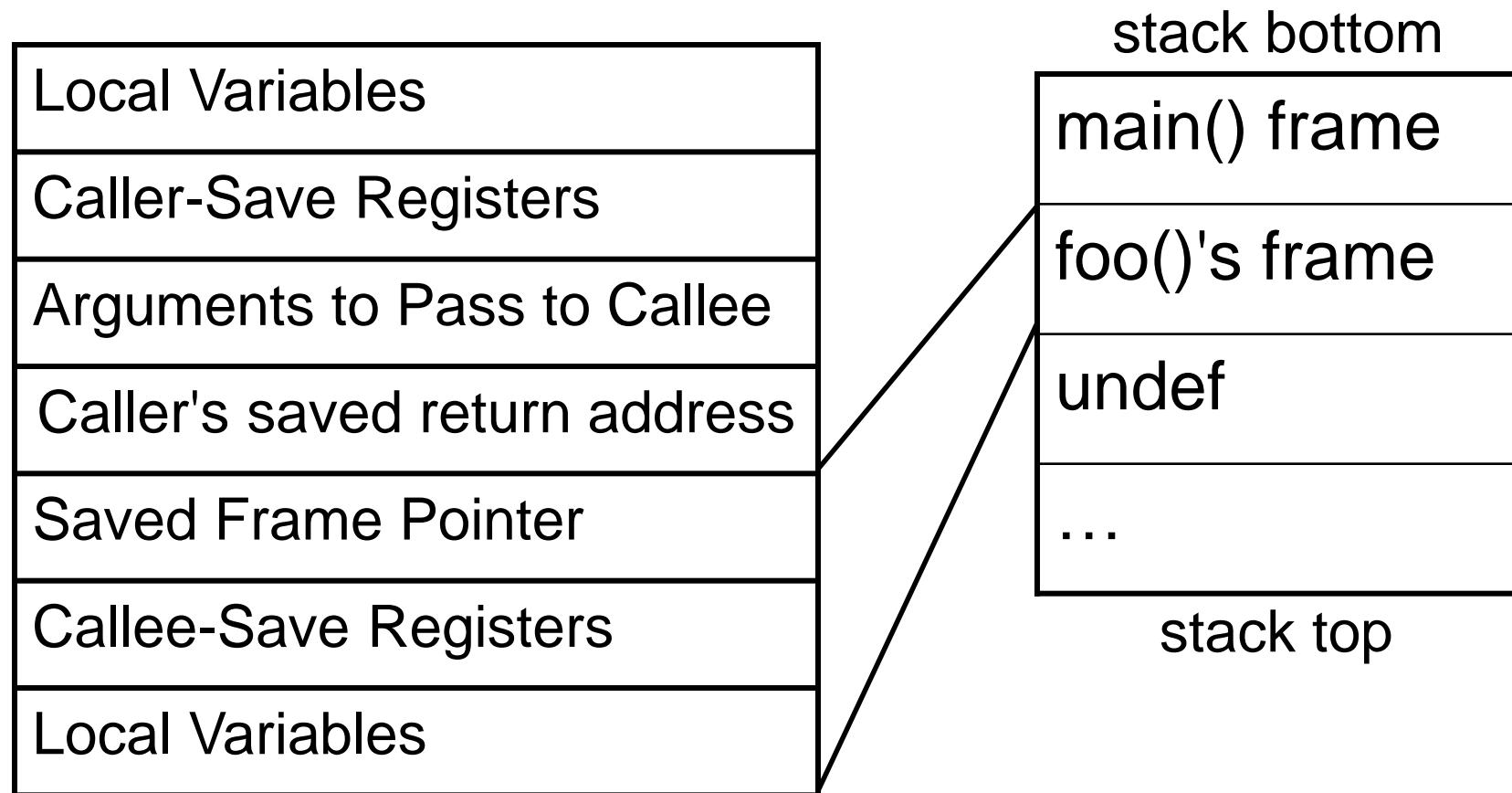
General Stack Frame Operation 4

When `foo()` starts, the frame pointer (`ebp`) still points to `main()`'s frame. So the first thing it does is to save the old frame pointer on the stack and set the new value to point to its own frame.



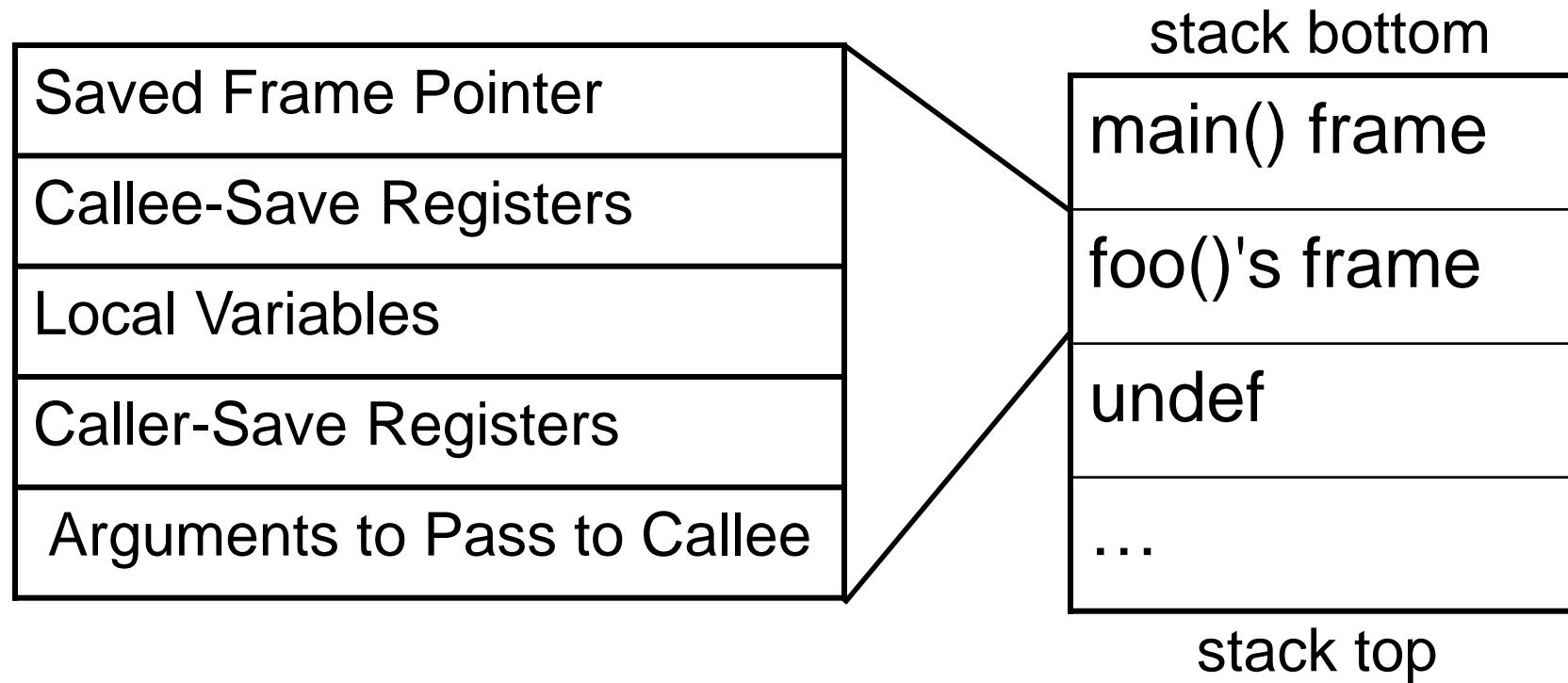
General Stack Frame Operation 5

Next, we'll assume the callee `foo()` would like to use all the registers, and must therefore save the callee-save registers. Then it will allocate space for its local variables.



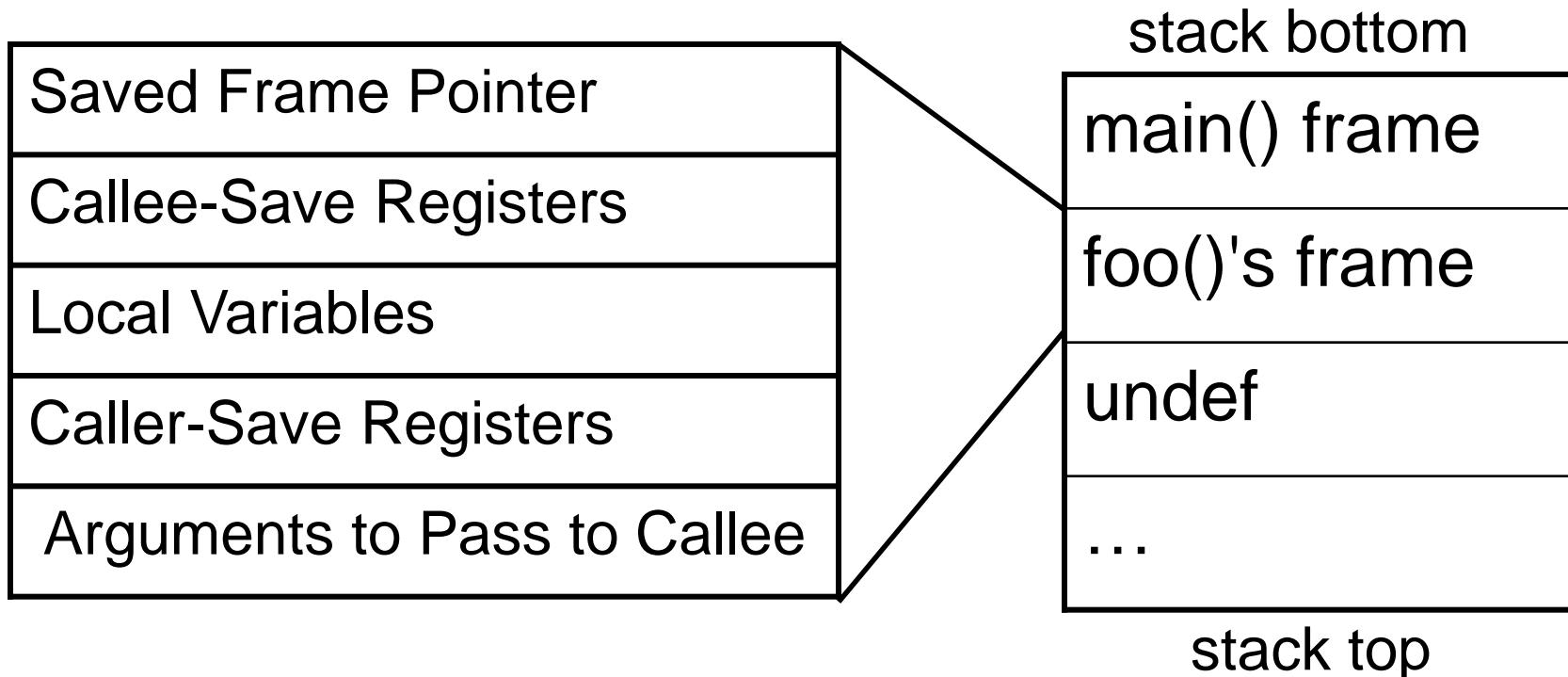
General Stack Frame Operation 6

At this point, `foo()` decides it wants to call `bar()`. It is still the callee-of-`main()`, but it will now be the caller-of-`bar`. So it saves any caller-save registers that it needs to. It then puts the function arguments on the stack as well.



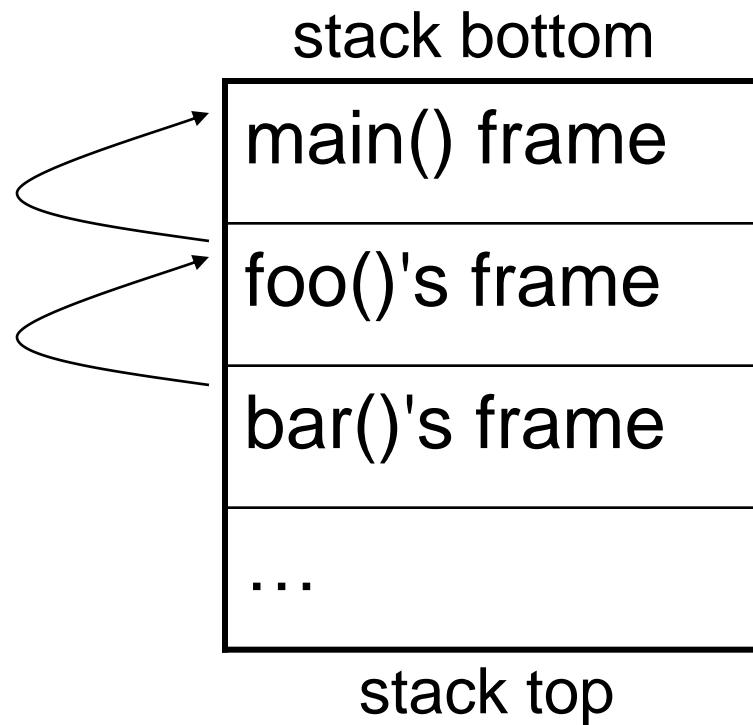
General Stack Frame Layout

Every part of the stack frame is technically optional (that is, you can hand code asm without following the conventions.) But compilers generate code which uses portions if they are needed. Which pieces are used can sometimes be manipulated with compiler options. (E.g. omit frame pointers, changing calling convention to pass arguments in registers, etc.)



Stack Frames are a Linked List!

The ebp in the current frame points at the saved ebp of the previous frame.



Example1.c

The stack frames in this example will be very simple.
Only saved frame pointer (ebp) and saved return addresses (eip).

```
//Example1 - using the stack
//to call subroutines
//New instructions:
//push, pop, call, ret, mov
int sub(){
    return 0xbeef;
}
int main(){
    sub();
    return 0xf00d;
}
```

	sub:	
00401000	push	ebp
00401001	mov	ebp,esp
00401003	mov	eax,0BEEFh
00401008	pop	ebp
00401009	ret	
	main:	
00401010	push	ebp
00401011	mov	ebp,esp
00401013	call	sub (401000h)
00401018	mov	eax,0F00Dh
0040101D	pop	ebp
0040101E	ret	

Example1.c 1:

EIP = 00401010, but no instruction yet executed

eax	0x003435C0 ☈
ebp	0x0012FFB8 ☈
esp	0x0012FF6C ☈

Key:

☒ **executed instruction**,

Ⓜ **modified value**

⌘ **start value**

sub:
00401000 push ebp
00401001 mov ebp,esp
00401003 mov eax,0BEEFh
00401008 pop ebp
00401009 ret
main:
00401010 push ebp
00401011 mov ebp,esp
00401013 call sub (401000h)
00401018 mov eax,0F00Dh
0040101D pop ebp
0040101E ret

Belongs to the
frame *before*
main() is called

0x0012FF6C
0x0012FF68
0x0012FF64
0x0012FF60
0x0012FF5C
0x0012FF58

0x004012E8 ☈
undef

Example1.c 2

eax	0x003435C0 ☈
ebp	0x0012FFB8 ☈
esp	0x0012FF68 ℗

sub:

00401000 push ebp
00401001 mov ebp,esp
00401003 mov eax,0BEEFh
00401008 pop ebp
00401009 ret

main:

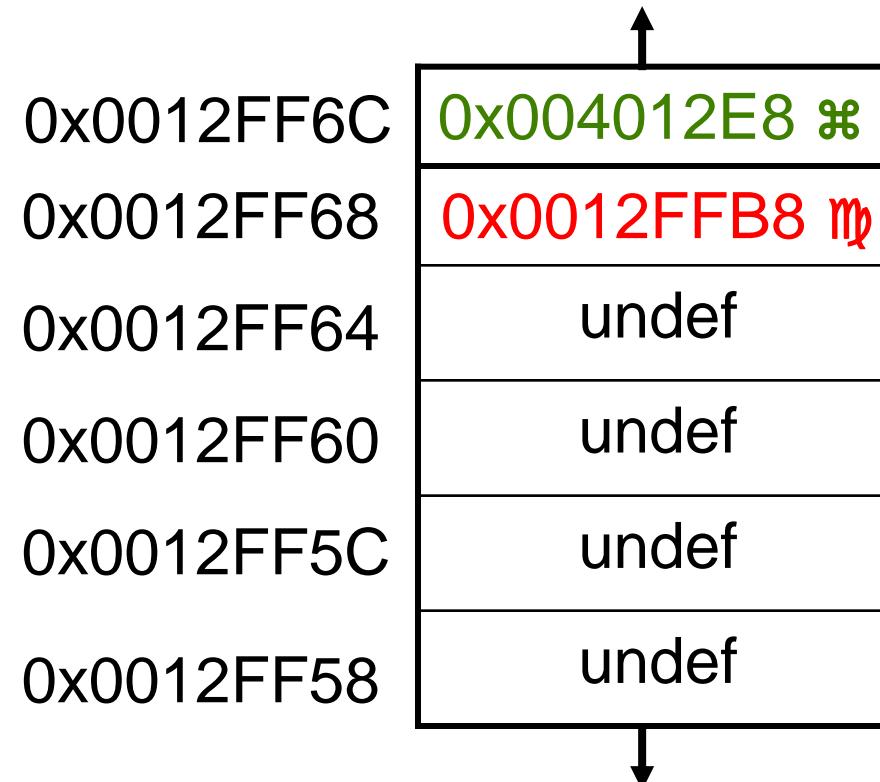
00401010 push ebp ☒
00401011 mov ebp,esp
00401013 call sub (401000h)
00401018 mov eax,0F00Dh
0040101D pop ebp
0040101E ret

Key:

☒ executed instruction,

℗ modified value

⌘ start value



Example1.c 3

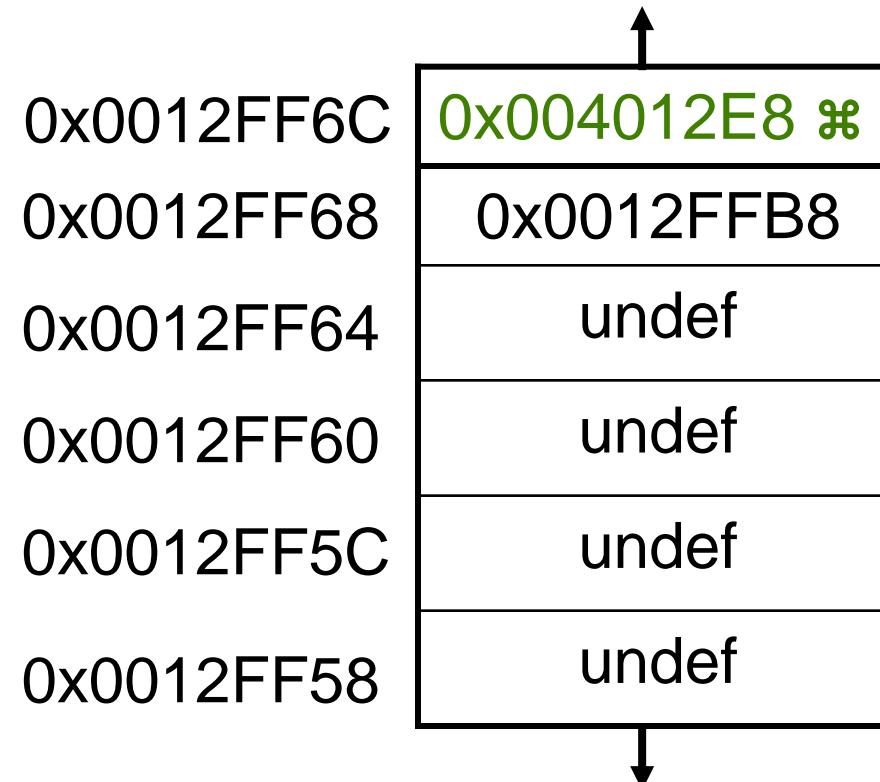
eax	0x003435C0	⌘
ebp	0x0012FF68	Ⓜ
esp	0x0012FF68	

Key:

- ☒ executed instruction,
- Ⓜ modified value
- ⌘ start value

sub:

```
00401000 push    ebp  
00401001 mov     ebp,esp  
00401003 mov     eax,0BEEFh  
00401008 pop    ebp  
00401009 ret  
  
main:  
00401010 push    ebp  
00401011 mov     ebp,esp ☒  
00401013 call    sub (401000h)  
00401018 mov     eax,0F00Dh  
0040101D pop    ebp  
0040101E ret
```



Example1.c 4

eax	0x003435C0
ebp	0x0012FF68
esp	0x0012FF64

Key:

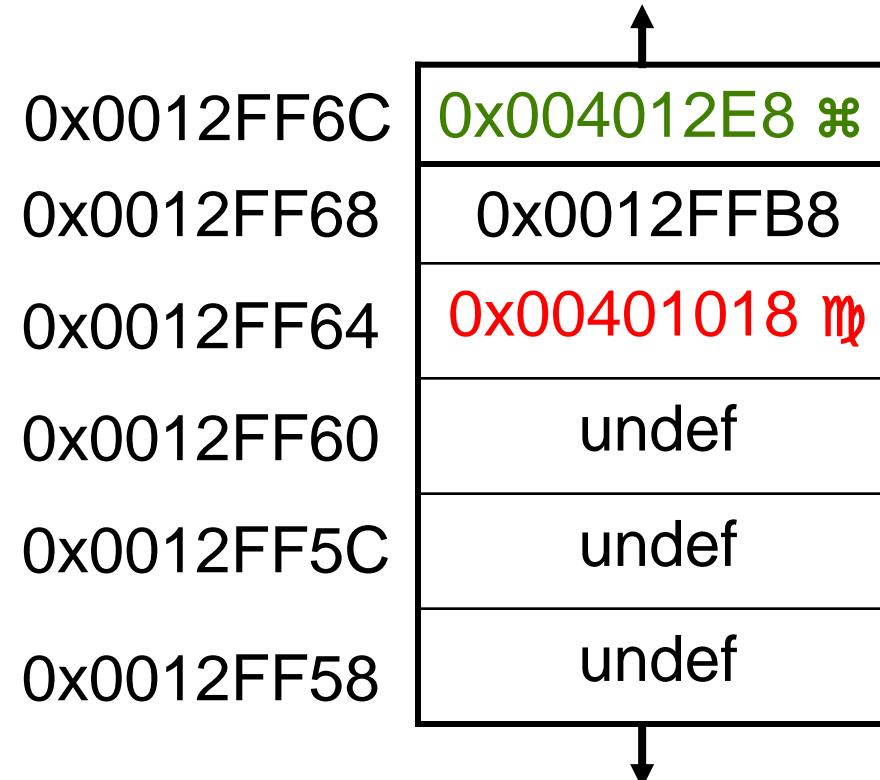
- ☒ executed instruction,
- Ⓜ modified value
- ⌘ start value

sub:

```
00401000 push    ebp
00401001 mov     ebp,esp
00401003 mov     eax,0BEEFh
00401008 pop     ebp
00401009 ret
```

main:

```
00401010 push    ebp
00401011 mov     ebp,esp
00401013 call    sub (401000h) ☒
00401018 mov     eax,0F00Dh
0040101D pop     ebp
0040101E ret
```



Example1.c 5

eax	0x003435C0 ☈
ebp	0x0012FF68
esp	0x0012FF60 ℗

Key:

- ☒ executed instruction,
- ℗ modified value
- ☈ start value

sub:

```
00401000 push    ebp ☒
00401001 mov     ebp,esp
00401003 mov     eax,0BEEFh
00401008 pop    ebp
00401009 ret
```

main:

```
00401010 push    ebp
00401011 mov     ebp,esp
00401013 call    sub (401000h)
00401018 mov     eax,0F00Dh
0040101D pop    ebp
0040101E ret
```

The diagram illustrates the state of the stack and memory. The stack frame is shown as a rectangle divided into four horizontal sections. The top section contains the value 0x004012E8 ☈. The second section contains 0x0012FFB8. The third section contains 0x00401018. The bottom section contains 0x0012FF68 ℗. Below the stack frame, a vertical list of memory addresses shows their values: 0x0012FF6C (0x004012E8 ☈), 0x0012FF68 (0x0012FFB8), 0x0012FF64 (0x00401018), 0x0012FF60 (0x0012FF68 ℗), 0x0012FF5C (undef), and 0x0012FF58 (undef). An upward arrow points from the esp register (0x0012FF60) to the stack frame, and a downward arrow points from the stack frame to the memory dump.

0x0012FF6C	0x004012E8 ☈
0x0012FF68	0x0012FFB8
0x0012FF64	0x00401018
0x0012FF60	0x0012FF68 ℗
0x0012FF5C	undef
0x0012FF58	undef

Example1.c 6

eax	0x003435C0	⌘
ebp	0x0012FF60	Ⓜ
esp	0x0012FF60	

Key:

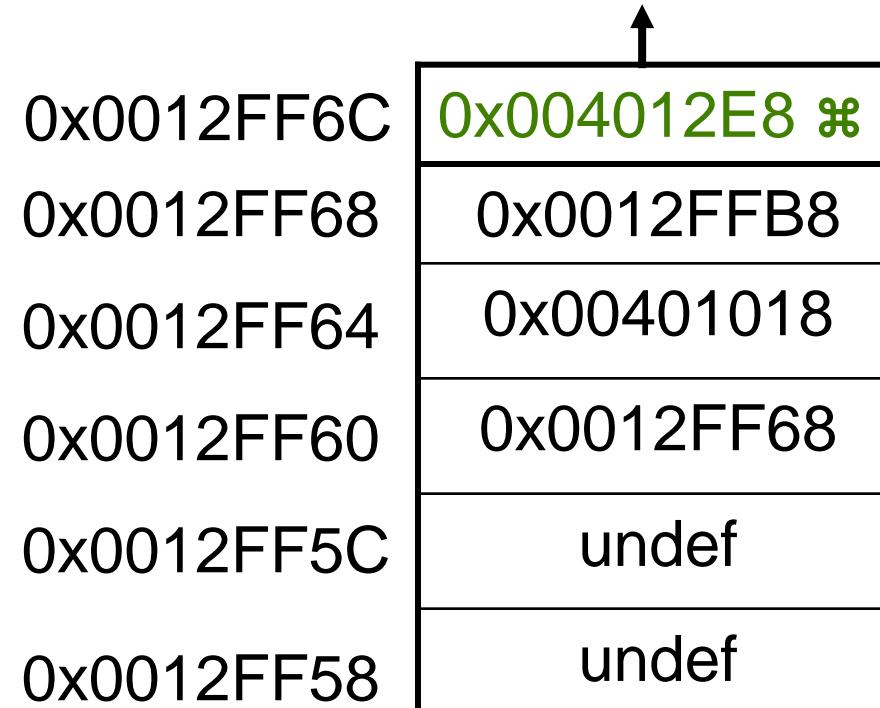
- ☒ executed instruction,
- Ⓜ modified value
- ⌘ start value

sub:

00401000 push ebp
00401001 mov ebp,esp ☒
00401003 mov eax,0BEEFh
00401008 pop ebp
00401009 ret

main:

00401010 push ebp
00401011 mov ebp,esp
00401013 call sub (401000h)
00401018 mov eax,0F00Dh
0040101D pop ebp
0040101E ret



Example1.c 6

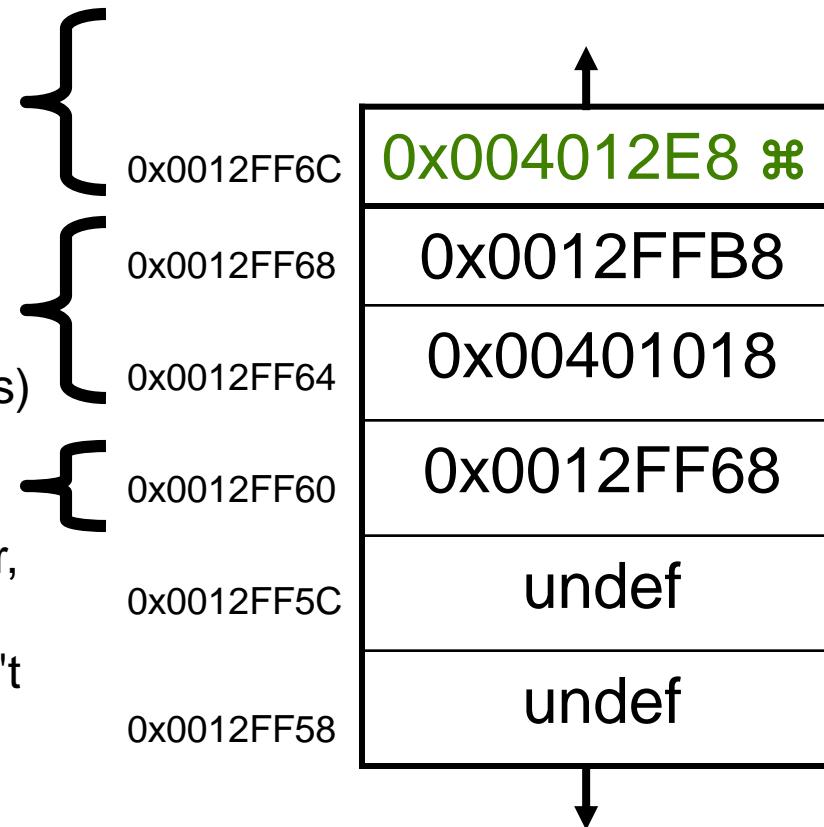
STACK FRAME TIME OUT

```
sub
push    ebp
mov    ebp, esp
mov    eax, 0BEEFh
pop    ebp
retn
main
push    ebp
mov    ebp, esp
call   _sub
mov    eax, 0F00Dh
pop    ebp
retn
```

“Function-before-main”’s frame

main’s frame
(saved frame pointer
and saved return address)

sub’s frame
(only saved frame pointer,
because it doesn’t call
anything else, and doesn’t
have local variables)



Example1.c 7

eax	0x0000BEEF
ebp	0x0012FF60
esp	0x0012FF60

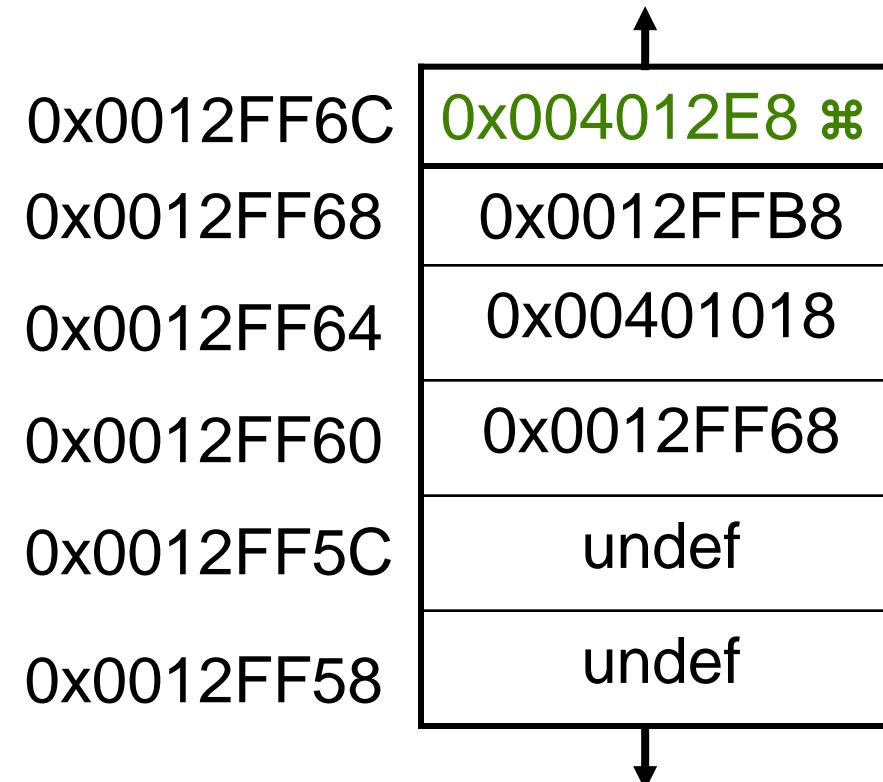
sub:

```
00401000 push    ebp
00401001 mov     ebp,esp
00401003 mov     eax,0BEEFh ✎
00401008 pop     ebp
00401009 ret

main:
00401010 push    ebp
00401011 mov     ebp,esp
00401013 call    sub (401000h)
00401018 mov     eax,0F00Dh
0040101D pop     ebp
0040101E ret
```

Key:

- ☒ executed instruction,
- Ⓜ modified value
- ⌘ start value



Example1.c 8

eax	0x0000BEEF
ebp	0x0012FF68 ℗
esp	0x0012FF64 ℗

sub:

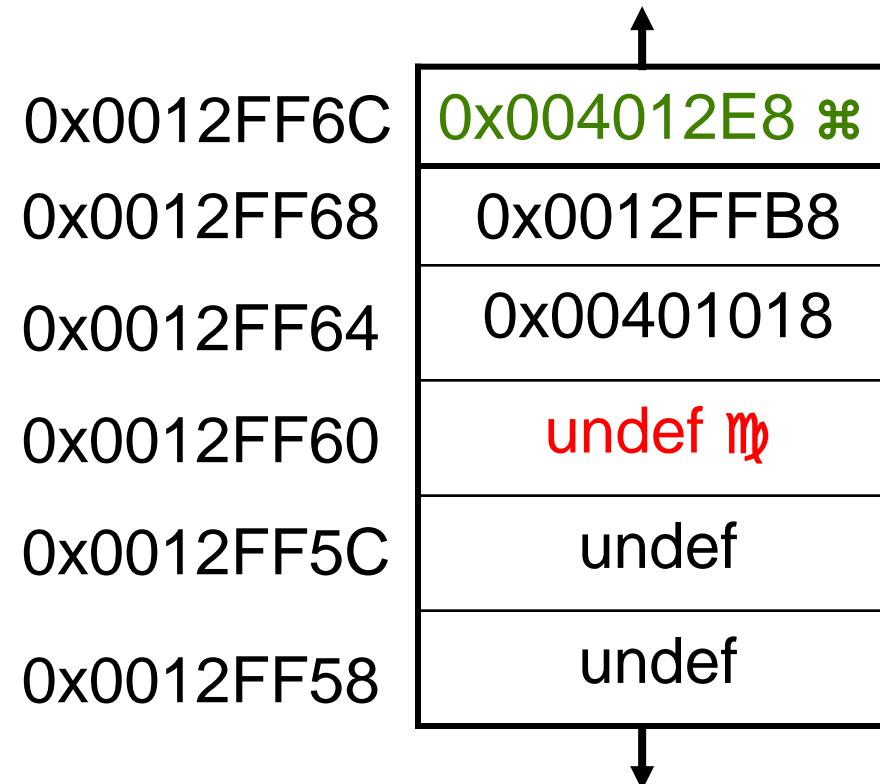
```
00401000 push    ebp
00401001 mov     ebp,esp
00401003 mov     eax,0BEEFh
00401008 pop    ebp ⊗
00401009 ret
```

main:

```
00401010 push    ebp
00401011 mov     ebp,esp
00401013 call    sub (401000h)
00401018 mov     eax,0F00Dh
0040101D pop    ebp
0040101E ret
```

Key:

- ⊗ executed instruction,
- ℗ modified value
- ⌘ start value



Example1.c 9

eax	0x0000BEEF
ebp	0x0012FF68
esp	0x0012FF68 

sub:

```
00401000 push    ebp
00401001 mov     ebp,esp
00401003 mov     eax,0BEEFh
00401008 pop    ebp
00401009 ret 
```

main:

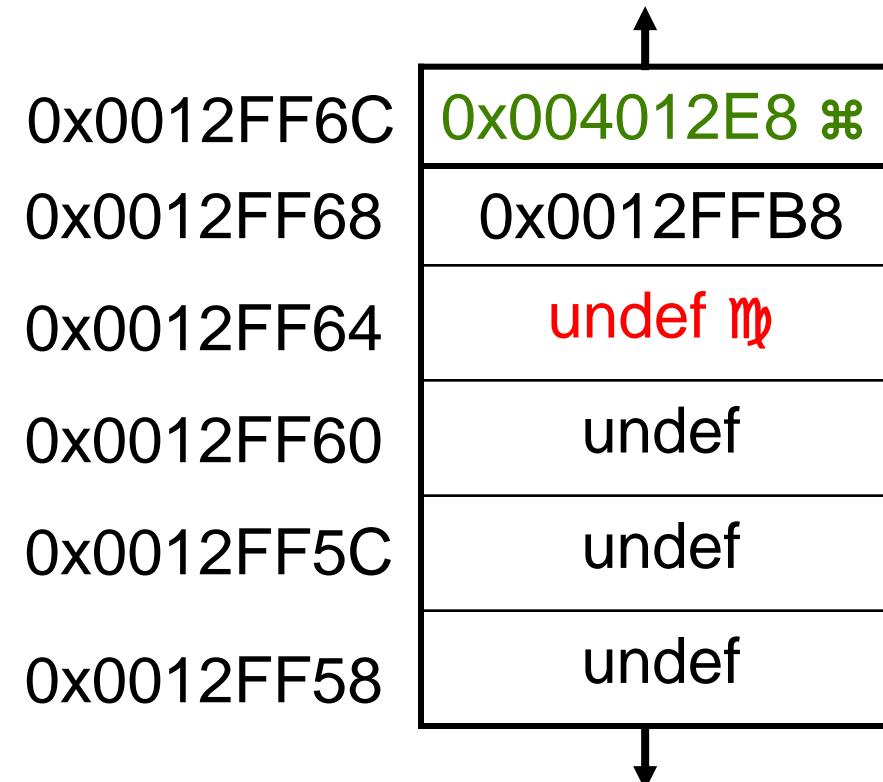
```
00401010 push    ebp
00401011 mov     ebp,esp
00401013 call    sub (401000h)
00401018 mov     eax,0F00Dh
0040101D pop    ebp
0040101E ret
```

Key:

 **executed instruction**,

 **modified value**

 **start value**



Example1.c 9

eax	0x0000F00D 
ebp	0x0012FF68
esp	0x0012FF68

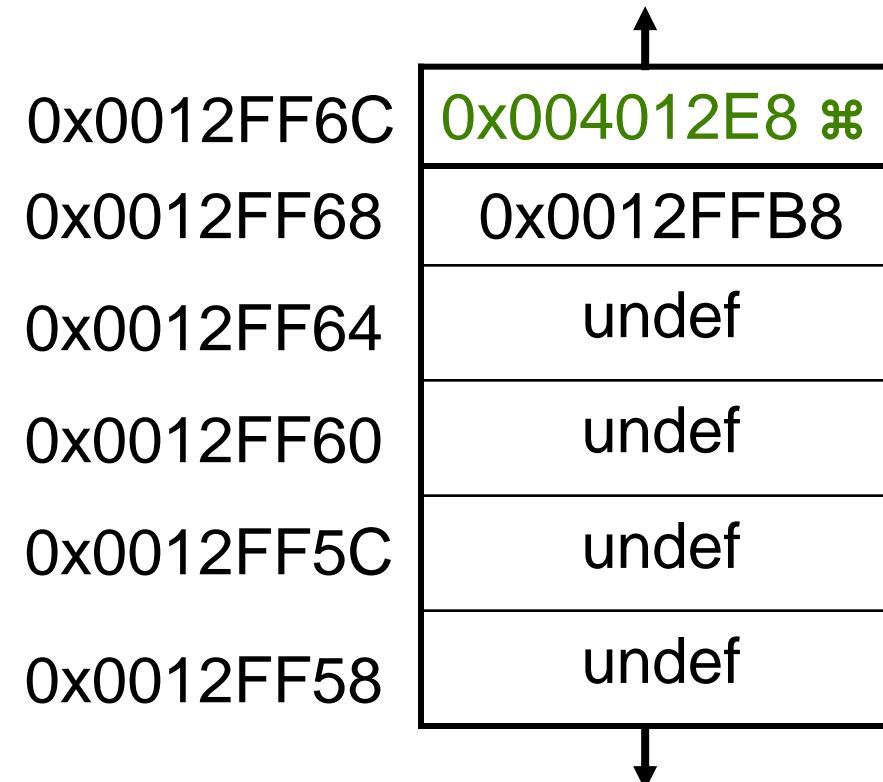
sub:

```
00401000 push    ebp
00401001 mov     ebp,esp
00401003 mov     eax,0BEEFh
00401008 pop     ebp
00401009 ret

main:
00401010 push    ebp
00401011 mov     ebp,esp
00401013 call    sub (401000h)
00401018 mov     eax,0F00Dh 
0040101D pop     ebp
0040101E ret
```

Key:

-  **executed instruction**,
-  **modified value**
-  **start value**



Example1.c 10

eax	0x0000F00D
ebp	0x0012FFB8 ℗
esp	0x0012FF6C ℗

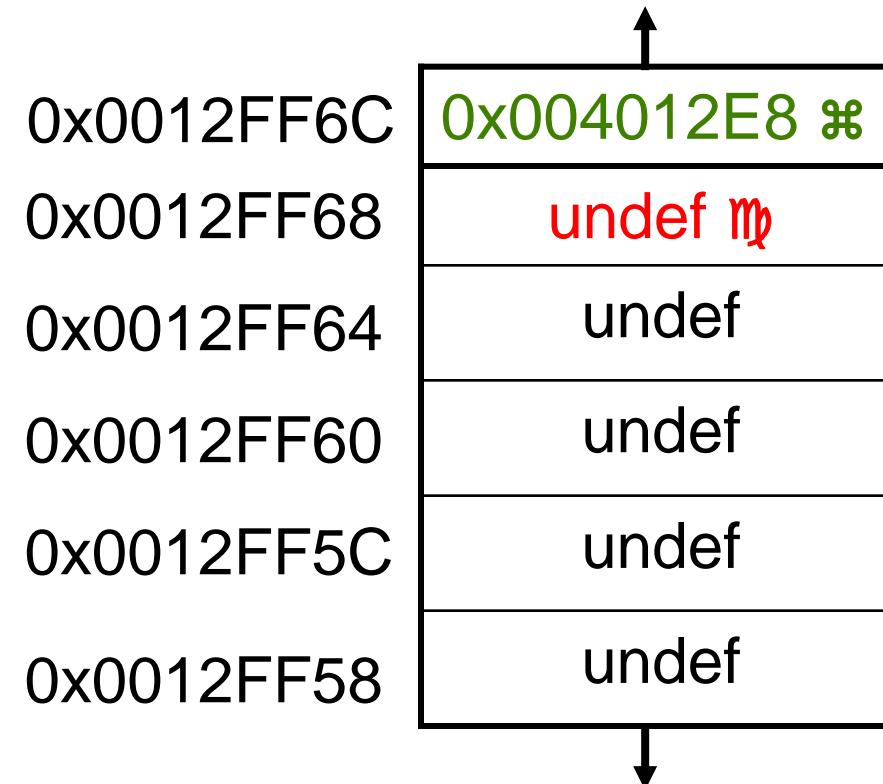
sub:

```
00401000 push    ebp
00401001 mov     ebp,esp
00401003 mov     eax,0BEEFh
00401008 pop     ebp
00401009 ret

main:
00401010 push    ebp
00401011 mov     ebp,esp
00401013 call    sub (401000h)
00401018 mov     eax,0F00Dh
0040101D pop     ebp ✎
0040101E ret
```

Key:

- ☒ executed instruction,
- ℗ modified value
- ⌘ start value



Example1.c 11

eax	0x0000F00D
ebp	0x0012FFB8
esp	0x0012FF70 

Key:

 **executed instruction**,

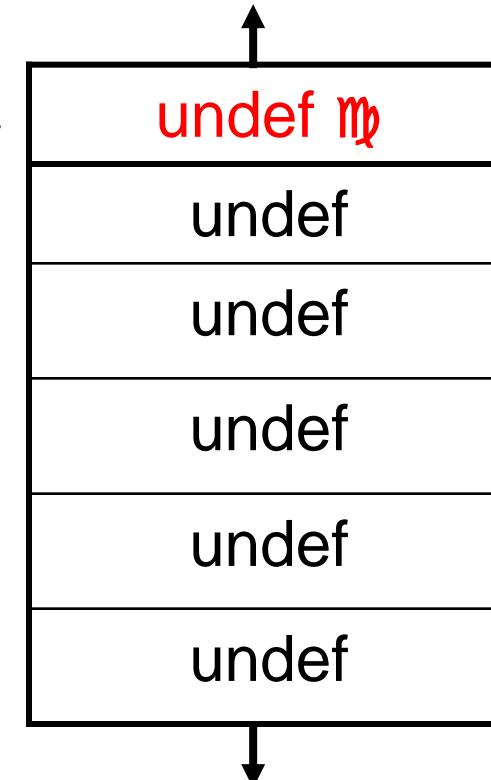
 **modified value**

 **start value**

sub:
00401000 push ebp
00401001 mov ebp,esp
00401003 mov eax,0BEEFh
00401008 pop ebp
00401009 ret

main:
00401010 push ebp
00401011 mov ebp,esp
00401013 call sub (401000h)
00401018 mov eax,0F00Dh
0040101D pop ebp
0040101E ret 

0x0012FF6C
0x0012FF68
0x0012FF64
0x0012FF60
0x0012FF5C
0x0012FF58



Execution would continue at the value ret removed from the stack: 0x004012E8

Example 1 Notes

- `sub()` is deadcode - its return value is not used for anything, and `main` always returns `0xF00D`. If optimizations are turned on in the compiler, it would remove `sub()`
- Because there are no input parameters to `sub()`, there is no difference whether we compile as `cdecl` vs `stdcall` calling conventions

Let's do that in a tool

- IMO, the best way to learn assembly is to write C code, compile it, and step through the assembly.
- Your assignment tonight will require you to do this. To keep the assembly code simple and free of confusing optimizations, you will have to change some settings in VS.
- For the sake of time, this has already been done on my demo machine.
- Let's use VS to look at the same code we just walked through.

"r/m32" Addressing Forms

- Anywhere you see an r/m32 it means it could be taking a value either from a register, or a memory address.
- I'm just calling these “r/m32 forms” because anywhere you see “r/m32” in the manual, the instruction can be a variation of the below forms.
- In Intel syntax, most of the time square brackets [] means to treat the value within as a memory address, and fetch the value at that address (like dereferencing a pointer)
 - mov eax, ebx
 - mov eax, [ebx]
 - mov eax, [ebx+ecx*X] (X=1, 2, 4, 8)
 - mov eax, [ebx+ecx*X+Y] (Y= one byte, 0-255 or 4 bytes, 0-2^32-1)
- Most complicated form is: [base + index*scale + disp]

LEA – Load Effective Address

- Frequently used with pointer arithmetic, sometimes for just arithmetic in general
- Uses the r/m32 form but **is the exception to the rule** that the square brackets [] syntax means dereference (“value at”)
 - **Remember this** – this tricks people up
- Example: ebx = 0x2, edx = 0x1000
 - lea eax, [edx+ebx*2]
 - eax = 0x1004, not the value at 0x1004

ADD and SUB

- Adds or Subtracts, just as expected
- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate
- No source ***and*** destination as r/m32s, because that could allow for memory to memory transfer, which isn't allowed on x86
- Evaluates the operation as if it were on signed AND unsigned data, and sets flags as appropriate. Instructions modify OF, SF, ZF, AF, PF, and CF flags
- add esp, 8
- sub eax, [ebx*2]

Example2.c - 1

```

.text:00000000 _sub:    push  ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
.text:00000009          lea    eax, [ecx+eax*2]
.text:0000000C          pop    ebp
.text:0000000D          retn
.text:00000010 _main:   push  ebp ☒
.text:00000011          mov    ebp, esp
.text:00000013          push   ecx
.text:00000014          mov    eax, [ebp+0Ch]
.text:00000017          mov    ecx, [eax+4]
.text:0000001A          push   ecx
.text:0000001B          call   dword ptr ds:_imp_atoi
.text:00000021          add    esp, 4
.text:00000024          mov    [ebp-4], eax
.text:00000027          mov    edx, [ebp-4]
.text:0000002A          push   edx
.text:0000002B          mov    eax, [ebp+8]
.text:0000002E          push   eax
.text:0000002F          call   _sub
.text:00000034          add    esp, 8
.text:00000037          mov    esp, ebp
.text:00000039          pop    ebp
.text:0000003A          retn

```

eax	0xcafe ⌘
ecx	0xbabe ⌘
edx	0xfeed ⌘
ebp	0x0012FF50 ⌘
esp	0x0012FF24 Ⓜ

0x0012FF30	0x12FFB0 (char ** argv) ⌘
0x0012FF2C	0x2 (int argc) ⌘
0x0012FF28	Addr after “call _main” ⌘
0x0012FF24	0x0012FF50 (saved ebp) Ⓜ
0x0012FF20	undef
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Key: **executed instruction ☒**, **modified value Ⓜ**, arbitrary example start value **⌘**

Example2.c - 2

```

.text:00000000 _sub:    push  ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
.text:00000009          lea    eax, [ecx+eax*2]
.text:0000000C          pop   ebp
.text:0000000D          retn
.text:00000010 _main:   push  ebp
.text:00000011          mov    ebp, esp ⊗
.text:00000013          push  ecx
.text:00000014          mov    eax, [ebp+0Ch]
.text:00000017          mov    ecx, [eax+4]
.text:0000001A          push  ecx
.text:0000001B          call   dword ptr ds:_imp_ atoi
.text:00000021          add    esp, 4
.text:00000024          mov    [ebp-4], eax
.text:00000027          mov    edx, [ebp-4]
.text:0000002A          push  edx
.text:0000002B          mov    eax, [ebp+8]
.text:0000002E          push  eax
.text:0000002F          call   _sub
.text:00000034          add    esp, 8
.text:00000037          mov    esp, ebp
.text:00000039          pop   ebp
.text:0000003A          retn

```

eax	0xcafe
ecx	0xbabe
edx	0xfeed
ebp	0x0012FF24 ↗
esp	0x0012FF24

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	undef
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2.c - 3

```
.text:00000000 _sub:    push  ebp  
.text:00000001          mov    ebp, esp  
.text:00000003          mov    eax, [ebp+8]  
.text:00000006          mov    ecx, [ebp+0Ch]  
.text:00000009          lea    eax, [ecx+eax*2]  
.text:0000000C          pop   ebp  
.text:0000000D          retn  
.text:00000010 _main:   push  ebp  
.text:00000011          mov    ebp, esp  
.text:00000013          push  ecx   
mov    eax, [ebp+0Ch]  
mov    ecx, [eax+4]  
push  ecx  
call  dword ptr ds:_imp_ atoi  
add   esp, 4  
mov    [ebp-4], eax  
mov    edx, [ebp-4]  
push  edx  
mov    eax, [ebp+8]  
push  eax  
call  _sub  
add   esp, 8  
mov    esp, ebp  
pop   ebp  
ret
```

Caller-save, or space for local var? This time it turns out to be space for local var since there is no corresponding pop, and the address is used later to refer to the value we know is stored in a.

eax	0xcafe
ecx	0xbabe
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a)
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2.c - 4

```

.text:00000000 _sub:    push  ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
.text:00000009          lea    eax, [ecx+eax*2]
.text:0000000C          pop   ebp
.text:0000000D          retn
.text:00000010 _main:   push  ebp
.text:00000011          mov    ebp, esp
.text:00000013          push   ecx
.text:00000014          mov    eax, [ebp+0Ch] █

        Getting the  
base of the  
argv char *  
array (aka  
argv[0])

        mov    ecx, [eax+4]
        push   ecx
        call   dword ptr ds:_imp_ atoi
        add    esp, 4
        mov    [ebp-4], eax
        mov    edx, [ebp-4]
        push   edx
        mov    eax, [ebp+8]
        push   eax
        call   _sub
        add    esp, 8
        mov    esp, ebp
        pop    ebp
        retn

```

eax	0x12FFB0 ¶
ecx	0xbabe
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a)
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 5

```
.text:00000000 _sub:    push  ebp  
.text:00000001          mov    ebp, esp  
.text:00000003          mov    eax, [ebp+8]  
.text:00000006          mov    ecx, [ebp+0Ch]  
.text:00000009          lea    eax, [ecx+eax*2]  
.text:0000000C          pop   ebp  
.text:0000000D          retn  
.text:00000010 _main:   push  ebp  
.text:00000011          mov    ebp, esp  
.text:00000013          push  ecx  
.text:00000014          mov    eax, [ebp+0Ch]  
.text:00000017          mov    ecx, [eax+4]   
push  ecx  
call  dword ptr ds:_imp_ atoi  
add   esp, 4  
mov   [ebp-4], eax  
mov   edx, [ebp-4]  
push  edx  
mov   eax, [ebp+8]  
push  eax  
call  _sub  
add   esp, 8  
mov   esp, ebp  
pop   ebp  
retn
```

Getting the
char * at
argv[1]
(I chose
0x12FFD4
arbitrarily since
it's out of the
stack scope
we're currently
looking at)

eax	0x12FFB0
ecx	0x12FFD4  (arbitrary #)
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after "call _main"
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a)
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 6

```

.text:00000000 _sub:
.text:00000001
.text:00000003
.text:00000006
.text:00000009

Saving some
slides...
This will push the
address of the
string at argv[1]
(0x12FFD4). atoi()
will read the string
and turn it into an
int, put that int in
eax, and return.
Then the adding 4
to esp will negate
the having pushed
the input parameter
and make
0x12FF1C
undefined again
(this is indicative of
cdecl)

```

push ebp
mov ebp, esp
mov eax, [ebp+8]
mov ecx, [ebp+0Ch]
lea eax, [ecx+eax*2]
pop ebp
retn
push ebp
mov ebp, esp
push ecx
mov eax, [ebp+0Ch]
mov ecx, [eax+4]
push ecx
call dword ptr ds:_imp_atoi
add esp, 4
mov [ebp-4], eax
mov edx, [ebp-4]
push edx
mov eax, [ebp+8]
push eax
call _sub
add esp, 8
mov esp, ebp
pop ebp
retn

eax	0x100M (arbitrary??)
ecx	0x12FFD4
edx	0xfeed
ebp	0x0012FF24
esp	0x0012FF20

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0xbabe (int a)
0x0012FF1C	undef M
0x0012FF18	undef M
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 7

```

.text:00000000 _sub:    push  ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
.text:00000009          lea    eax, [ecx+eax*2]
.text:0000000C          pop    ebp
.text:0000000D          retn
.text:00000010 _main:   push  ebp
.text:00000011          mov    ebp, esp
.text:00000013          push  ecx
                           mov    eax, [ebp+0Ch]
                           mov    ecx, [eax+4]
                           push  ecx
                           call   dword ptr ds:_imp_atoi
                           add    esp, 4
                           mov    [ebp-4], eax ☒
                           mov    edx, [ebp-4] ☒
                           push  edx ☒
                           mov    eax, [ebp+8]
                           push  eax
                           call   _sub
                           add    esp, 8
                           mov    esp, ebp
                           pop    ebp
                           retn

```

First setting “a” equal to the return value. Then pushing “a” as the second parameter in sub(). We can see an obvious optimization would have been to replace the last two instructions with “push eax”.

eax	0x100
ecx	0x12FFD4
edx	0x100 ☒
ebp	0x0012FF24
esp	0x0012FF1C ☒

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 ☒ (int a) ☒
0x0012FF1C	0x100 ☒ (int y) ☒
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 8

```

.text:00000000 _sub:    push  ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
.text:00000009          lea    eax, [ecx+eax*2]
.text:0000000C          pop   ebp
.text:0000000D          retn
.text:00000010 _main:   push  ebp
.text:00000011          mov    ebp, esp
.text:00000013          push   ecx
.text:00000014          mov    eax, [ebp+0Ch]
.text:00000017          mov    ecx, [eax+4]
.text:0000001A          push   ecx
.text:0000001B          call   dword ptr ds:_imp_ atoi
.text:00000021          add    esp, 4
.text:00000024          mov    [ebp-4], eax
.text:00000027          mov    edx, [ebp-4]
.text:0000002A          push   edx
mov    eax, [ebp+8] ⊗
push   eax ⊗
call   _sub
add    esp, 8
mov    esp, ebp
pop    ebp
retn

```

Pushing argc as the first parameter (int x) to sub()

eax	0x2 ™
ecx	0x12FFD4
edx	0x100
ebp	0x0012FF24
esp	0x0012FF18 ™

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x) ™
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 9

```

.text:00000000 _sub:    push  ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
.text:00000009          lea    eax, [ecx+eax*2]
.text:0000000C          pop   ebp
.text:0000000D          retn
.text:00000010 _main:   push  ebp
.text:00000011          mov    ebp, esp
.text:00000013          push   ecx
.text:00000014          mov    eax, [ebp+0Ch]
.text:00000017          mov    ecx, [eax+4]
.text:0000001A          push   ecx
.text:0000001B          call   dword ptr ds:_imp_ atoi
.text:00000021          add    esp, 4
.text:00000024          mov    [ebp-4], eax
.text:00000027          mov    edx, [ebp-4]
.text:0000002A          push   edx
.text:0000002B          mov    eax, [ebp+8]
.text:0000002E          push   eax
.text:0000002F          call   _sub ☒
.text:00000034          add    esp, 8
.text:00000037          mov    esp, ebp
.text:00000039          pop    ebp
.text:0000003A          retn

```

eax	0x2
ecx	0x12FFD4
edx	0x100
ebp	0x0012FF24
esp	0x0012FF14 m

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034 m
0x0012FF10	undef
0x0012FF0C	undef



Example2 - 10

```

.text:00000000 _sub:    push  ebp ⊗
.text:00000001          mov   ebp, esp ⊗
.text:00000003          mov   eax, [ebp+8]
.text:00000006          mov   ecx, [ebp+0Ch]
.text:00000009          lea   eax, [ecx+eax*2]
.text:0000000C          pop   ebp
.text:0000000D          retn
.text:00000010 _main:   push  ebp
.text:00000011          mov   ebp, esp
.text:00000013          push  ecx
.text:00000014          mov   eax, [ebp+0Ch]
.text:00000017          mov   ecx, [eax+4]
.text:0000001A          push  ecx
.text:0000001B          call  dword ptr ds:_imp_atoi
.text:00000021          add   esp, 4
.text:00000024          mov   [ebp-4], eax
.text:00000027          mov   edx, [ebp-4]
.text:0000002A          push  edx
.text:0000002B          mov   eax, [ebp+8]
.text:0000002E          push  eax
.text:0000002F          call  _sub
.text:00000034          add   esp, 8
.text:00000037          mov   esp, ebp
.text:00000039          pop   ebp
.text:0000003A          retn

```

eax	0x2
ecx	0x12FFD4
edx	0x100
ebp	0x0012FF10 ↗
esp	0x0012FF10 ↗

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034
0x0012FF10	0x0012FF24(saved ebp) ↗
0x0012FF0C	undef

Example2 - 11

```

.text:00000000 _sub:
.text:00000001
    push    ebp
    mov     ebp, esp
    mov     eax, [ebp+8] ⊗
    mov     ecx, [ebp+0Ch] ⊗
    lea     eax, [ecx+eax*2]
    pop    ebp
    retn
.text:00000010 _main:
.text:00000011
.text:00000013
.push    ecx
.text:00000014
    mov     eax, [ebp+0Ch]
.text:00000017
    mov     ecx, [eax+4]
.push    ecx
.text:0000001B
    call   dword ptr ds:_imp_atoi
.add    esp, 4
.text:00000021
    mov     [ebp-4], eax
.text:00000024
    mov     edx, [ebp-4]
.push    edx
.text:0000002B
    mov     eax, [ebp+8]
.text:0000002E
.push    eax
.call   _sub
.text:0000002F
.add    esp, 8
.text:00000034
    mov     esp, ebp
.pop    ebp
.retn
.text:00000039
.text:0000003A

```

Move “x” into eax, and “y” into ecx.

eax	0x2 ⊗ (no value change)
ecx	0x100 ⊗
edx	0x100
ebp	0x0012FF10
esp	0x0012FF10

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034
0x0012FF10	0x0012FF24 (saved ebp)
0x0012FF0C	undef

Example2 - 12

```

.text:00000000 _sub:
.text:00000001
.text:00000003
.text:00000005
.text:00000007
Set the return value
(eax) to 2*x + y.
Note: neither
pointer arith, nor an
“address” which
was loaded. Just an
efficient way to do a
calculation.
.text:0000001B
.text:00000021
.text:00000024
.text:00000027
.text:0000002A
.text:0000002B
.text:0000002E
.text:0000002F
.text:00000034
.text:00000037
.text:00000039
.text:0000003A
    push    ebp
    mov     ebp, esp
    mov     eax, [ebp+8]
    mov     ecx, [ebp+0Ch]
    lea     eax, [ecx+eax*2] ⊗
    pop     ebp
    retn
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     eax, [ebp+0Ch]
    mov     ecx, [eax+4]
    push    ecx
    call    dword ptr ds:_imp_atoi
    add    esp, 4
    mov     [ebp-4], eax
    mov     edx, [ebp-4]
    push    edx
    mov     eax, [ebp+8]
    push    eax
    call    _sub
    add    esp, 8
    mov     esp, ebp
    pop     ebp
    retn

```

eax	0x104 ↗
ecx	0x100
edx	0x100
ebp	0x0012FF10
esp	0x0012FF10

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034
0x0012FF10	0x0012FF24 (saved ebp)
0x0012FF0C	undef

Example2 - 13

```

.text:00000000 _sub:    push  ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
.text:00000009          lea    eax, [ecx+eax*2]
.text:0000000C          pop    ebp ⊗
.text:0000000D          retn
.text:00000010 _main:   push  ebp
.text:00000011          mov    ebp, esp
.text:00000013          push  ecx
.text:00000014          mov    eax, [ebp+0Ch]
.text:00000017          mov    ecx, [eax+4]
.text:0000001A          push  ecx
.text:0000001B          call   dword ptr ds:_imp_ atoi
.text:00000021          add    esp, 4
.text:00000024          mov    [ebp-4], eax
.text:00000027          mov    edx, [ebp-4]
.text:0000002A          push  edx
.text:0000002B          mov    eax, [ebp+8]
.text:0000002E          push  eax
.text:0000002F          call   _sub
.text:00000034          add    esp, 8
.text:00000037          mov    esp, ebp
.text:00000039          pop    ebp
.text:0000003A          retn

```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24 ↗
esp	0x0012FF14 ↗

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	0x00000034
0x0012FF10	undef ↗
0x0012FF0C	undef

Example2 - 14

```

.text:00000000 _sub:    push  ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
.text:00000009          lea    eax, [ecx+eax*2]
.text:0000000C          pop    ebp
.text:0000000D          retn  ☒
.text:00000010 _main:   push  ebp
.text:00000011          mov    ebp, esp
.text:00000013          push   ecx
.text:00000014          mov    eax, [ebp+0Ch]
.text:00000017          mov    ecx, [eax+4]
.text:0000001A          push   ecx
.text:0000001B          call   dword ptr ds:_imp_ atoi
.text:00000021          add    esp, 4
.text:00000024          mov    [ebp-4], eax
.text:00000027          mov    edx, [ebp-4]
.text:0000002A          push   edx
.text:0000002B          mov    eax, [ebp+8]
.text:0000002E          push   eax
.text:0000002F          call   _sub
.text:00000034          add    esp, 8
.text:00000037          mov    esp, ebp
.text:00000039          pop    ebp
.text:0000003A          retn

```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24
esp	0x0012FF18 Ⓜ

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	0x100 (int y)
0x0012FF18	0x2 (int x)
0x0012FF14	undef Ⓜ
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 15

```

.text:00000000 _sub:    push  ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
.text:00000009          lea    eax, [ecx+eax*2]
.text:0000000C          pop    ebp
.text:0000000D          retn
.text:00000010 _main:   push  ebp
.text:00000011          mov    ebp, esp
.text:00000013          push  ecx
.text:00000014          mov    eax, [ebp+0Ch]
.text:00000017          mov    ecx, [eax+4]
.text:0000001A          push  ecx
.text:0000001B          call   dword ptr ds:_imp_ atoi
.text:00000021          add    esp, 4
.text:00000024          mov    [ebp-4], eax
.text:00000027          mov    edx, [ebp-4]
.text:0000002A          push  edx
.text:0000002B          mov    eax, [ebp+8]
.text:0000002E          push  eax
.text:0000002F          call   _sub
.btext:00000034          add    esp, 8 ⊞
.text:00000037          mov    esp, ebp
.text:00000039          pop    ebp
.text:0000003A          retn

```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24
esp	0x0012FF20 ↗

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	0x100 (int a)
0x0012FF1C	undef ↗
0x0012FF18	undef ↗
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 16

```

.text:00000000 _sub:    push  ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
.text:00000009          lea    eax, [ecx+eax*2]
.text:0000000C          pop    ebp
.text:0000000D          retn
.text:00000010 _main:   push  ebp
.text:00000011          mov    ebp, esp
.text:00000013          push  ecx
.text:00000014          mov    eax, [ebp+0Ch]
.text:00000017          mov    ecx, [eax+4]
.text:0000001A          push  ecx
.text:0000001B          call   dword ptr ds:_imp_ atoi
.text:00000021          add    esp, 4
.text:00000024          mov    [ebp-4], eax
.text:00000027          mov    edx, [ebp-4]
.text:0000002A          push  edx
.text:0000002B          mov    eax, [ebp+8]
.text:0000002E          push  eax
.text:0000002F          call   _sub
.text:00000034          add    esp, 8
.text:00000037          mov    esp, ebp ⊗
.text:00000039          pop    ebp
.text:0000003A          retn

```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF24
esp	0x0012FF24 ↗

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	0x0012FF50 (saved ebp)
0x0012FF20	undef ↗
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef

Example2 - 17

```

.text:00000000 _sub:    push  ebp
.text:00000001          mov    ebp, esp
.text:00000003          mov    eax, [ebp+8]
.text:00000006          mov    ecx, [ebp+0Ch]
.text:00000009          lea    eax, [ecx+eax*2]
.text:0000000C          pop   ebp
.text:0000000D          retn
.text:00000010 _main:   push  ebp
.text:00000011          mov    ebp, esp
.text:00000013          push   ecx
.text:00000014          mov    eax, [ebp+0Ch]
.text:00000017          mov    ecx, [eax+4]
.text:0000001A          push   ecx
.text:0000001B          call   dword ptr ds:_imp_ atoi
.text:00000021          add    esp, 4
.text:00000024          mov    [ebp-4], eax
.text:00000027          mov    edx, [ebp-4]
.text:0000002A          push   edx
.text:0000002B          mov    eax, [ebp+8]
.text:0000002E          push   eax
.text:0000002F          call   _sub
.text:00000034          add    esp, 8
.text:00000037          mov    esp, ebp
.text:00000039          pop    ebp ⊗
.text:0000003A          retn

```

eax	0x104
ecx	0x100
edx	0x100
ebp	0x0012FF50 ↗
esp	0x0012FF28 ↗

0x0012FF30	0x12FFB0 (char ** argv)
0x0012FF2C	0x2 (int argc)
0x0012FF28	Addr after “call _main”
0x0012FF24	undef ↗
0x0012FF20	undef
0x0012FF1C	undef
0x0012FF18	undef
0x0012FF14	undef
0x0012FF10	undef
0x0012FF0C	undef



Example 2 In VS

JMP

- Change eip to the given address
- Main forms of the address
 - Short relative (1 byte displacement from end of the instruction)
 - “jmp 00401023” doesn’t have the number 00401023 anywhere in it, it’s really “jmp 0xE bytes forward”
 - Some disassemblers will indicate this with a mnemonic by writing it as “jmp short”
 - Near relative (4 byte displacement from current eip)
 - Absolute (hardcoded address in instruction)
 - Absolute Indirect (address calculated with r/m32)
- jmp -2 == infinite loop for short relative jmp
 - Can be useful in patching a hard breakpoint.

Example3.c

(Remain calm)

```
int main(){
    int a=1, b=2;
    if(a == b){
        return 1;
    }
    if(a > b){
        return 2;
    }
    if(a < b){
        return 3;
    }
    return 0xdefea7;
}
```

main:

00401010 push	ebp
00401011 mov	ebp,esp
00401013 sub	esp,8
00401016 mov	dword ptr [ebp-4],1
0040101D mov	dword ptr [ebp-8],2
00401024 mov	eax,dword ptr [ebp-4]
00401027 cmp	eax,dword ptr [ebp-8]
0040102A jne	00401033
0040102C mov	eax,1
00401031 jmp	00401056
00401033 mov	ecx,dword ptr [ebp-4]
00401036 cmp	ecx,dword ptr [ebp-8]
00401039 jle	00401042
0040103B mov	eax,2
00401040 jmp	00401056
00401042 mov	edx,dword ptr [ebp-4]
00401045 cmp	edx,dword ptr [ebp-8]
00401048 jge	00401051
0040104A mov	eax,3
0040104F jmp	00401056
00401051 mov	eax,0DEFEA7h
00401056 mov	esp,ebp
00401058 pop	ebp
00401059 ret	

Jcc {



JCC – Jump if Condition is Met

- There are more than 4 pages of conditional jump types! Luckily a bunch of them are synonyms for each other.
- For example: JNE == JNZ (Jump if not equal, Jump if not zero, both check if the Zero Flag (ZF) == 0)

JCC Instruction Quick Reference

- JZ/JE: if ZF == 1
- JNZ/JNE: if ZF == 0
- JLE/JNG : if ZF == 1 or SF != OF
- JGE/JNL : if SF == OF
- JBE: if CF == 1 OR ZF == 1
- JB: if CF == 1
- Note: Don't get hung up on memorizing which flags are set for what. More often than not, you will be running code in a debugger, not just reading it. In the debugger you can just look at eflags and/or look for the blinking line in Ida.
 - You can also always google or reference this slide

What Sets the Flags?

- Before you can do a conditional jump, you need something to set the condition flags for you.
- Typically done with CMP, TEST, or whatever instructions are already inline and happen to have flag-setting side-effects

CMP – Compare Two Operands

- “The comparison is performed by subtracting the second operand from the first operand and then setting the status flags in the same manner as the SUB instruction.”
- What’s the difference from just doing SUB?
- Modifies CF, OF, SF, ZF, AF, and PF
- (implies that SUB modifies all those too)

TEST – Logical (bit-wise) Compare

- “Computes the bit-wise logical AND of first operand (source 1 operand) and the second operand (source 2 operand) and sets the SF, ZF, and PF status flags according to the result.”
- Like CMP - sets flags, and throws away the result

AND – Logical AND

- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate (No source *and* destination as r/m32s)

OR – Logical Inclusive OR

- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate (*No source and destination as r/m32s*)

XOR – Logical Exclusive OR

- Destination operand can be r/m32 or register
- Source operand can be r/m32 or register or immediate (No source *and* destination as r/m32s)
- Why might you see the instruction “xor eax eax”

NOT – One's Complement Negation

- Single source/destination operand can be r/m32

Example5.c - simple for loop

```
#include <stdio.h>

int main(){
    int i;
    for(i = 0; i < 10; i++){
        printf("i = %d\n", i);
    }
}
```

What does this add say
about the calling
convention of printf()?

Interesting note:
Defaults to returning 0

main:

00401010	push	ebp
00401011	mov	ebp,esp
00401013	push	ecx
00401014	mov	dword ptr [ebp-4],0
0040101B	jmp	00401026
0040101D	mov	eax,dword ptr [ebp-4]
00401020	add	eax,1
00401023	mov	dword ptr [ebp-4],eax
00401026	cmp	dword ptr [ebp-4],0Ah
0040102A	jge	00401040
0040102C	mov	ecx,dword ptr [ebp-4]
0040102F	push	ecx
00401030	push	405000h
00401035	call	dword ptr ds:[00406230h]
0040103B	add	esp,8
0040103E	jmp	0040101D
00401040	xor	eax,eax
00401042	mov	esp,ebp
00401044	pop	ebp
00401045	ret	

SHL – Shift Logical Left

- Can be explicitly used with the C “`<<`” operator
- First operand (source and destination) operand is an r/m32
- Second operand is either cl (lowest byte of ecx), or a 1 byte immediate. The 2nd operand is the number of places to shift.
- It **multiplies** the register by 2 for each place the value is shifted. More efficient than a multiply instruction.
 - Often a piece of compiler optimization
- Bits shifted off the left hand side are “shifted into” (set) the carry flag (CF) – all that matters is the last bit shifted

`shl cl, 2`

	00110011b (cl - 0x33)
result	11001100b (cl - 0xCC) CF = 0

`shl cl, 3`

	00110011b (cl - 0x33)
result	10011000b (cl - 0x98) CF = 1

SHR – Shift Logical Right

- Can be explicitly used with the C “`>>`” operator
- First operand (source and destination) operand is an r/m32
- Second operand is either cl (lowest byte of ecx), or a 1 byte immediate. The 2nd operand is the number of places to shift.
- It **divides** the register by 2 for each place the value is shifted. More efficient than a multiply instruction.
- Bits shifted off the right hand side are “shifted into” (set) the carry flag (CF) – all that matters is the last bit shifted

`shr cl, 2`

	00110011b (cl - 0x33)
result	00001100b (cl - 0x0C) CF = 1

`shr cl, 3`

	00110011b (cl - 0x33)
result	00000110b (cl - 0x06) CF = 0

Example6.c

```
//Multiply and divide transformations  
//New instructions:  
//shl - Shift Left, shr - Shift Right  
  
int main(){  
    unsigned int a, b, c;  
    a = 0x40;  
    b = a * 8;  
    c = b / 16;  
    return c;  
}
```

	main:	
push	ebp	
mov	ebp,esp	
sub	esp,0Ch	
mov	dword ptr [ebp-4],40h	
mov	eax,dword ptr [ebp-4]	
★ shl	eax,3	
mov	dword ptr [ebp-8],eax	
mov	ecx,dword ptr [ebp-8]	
★ shr	ecx,4	
mov	dword ptr [ebp-0Ch],ecx	
mov	eax,dword ptr [ebp-0Ch]	
mov	esp,ebp	
pop	ebp	
ret		

IMUL – Signed Multiply

- Wait...what? Weren't the operands unsigned?
 - Visual Studio seems to have a predilection for imul over mul (unsigned multiply). I haven't been able to get it to generate the latter for simple examples.
 - Three forms. One, two, or three operands
 - imul r/m32 $\text{edx:eax} = \text{eax} * \text{r/m32}$
 - imul reg, r/m32 $\text{reg} = \text{reg} * \text{r/m32}$
 - $\text{imul reg, r/m32, immediate}$ $\text{reg} = \text{r/m32} * \text{immediate}$

initial	<table border="1"><tr><td>edx</td><td>eax</td><td>r/m32(ecx)</td></tr><tr><td>0x0</td><td>0x44000000</td><td>0x4</td></tr></table>	edx	eax	r/m32(ecx)	0x0	0x44000000	0x4
edx	eax	r/m32(ecx)					
0x0	0x44000000	0x4					
operation	imul ecx						
result	<table border="1"><tr><td>edx</td><td>eax</td><td>r/m32(ecx)</td></tr><tr><td>0x1</td><td>0x10000000</td><td>0x4</td></tr></table>	edx	eax	r/m32(ecx)	0x1	0x10000000	0x4
edx	eax	r/m32(ecx)					
0x1	0x10000000	0x4					

eax	r/m32(ecx)
0x20	0x4

eax	r/m32(ecx)
0x20	0x4

imul eax, ecx, 0x6	
eax	r/m32(ecx)
0x18	0x4

DIV – Unsigned Divide

- Two forms
 - Unsigned divide ax by r/m8, al = quotient, ah = remainder
 - Unsigned divide edx:eax by r/m32, eax = quotient, edx = remainder
- If dividend is 32bits, edx will just be set to 0 before the instruction (as occurred in the Example7.c code)
- If the divisor is 0, a divide by zero exception is raised.

initial
↓
operation
↓
result

ax	r/m8(cx)
0x8	0x3

div ax, cx

ah	al
0x2	0x2

edx	eax	r/m32(ecx)
0x0	0x8	0x3

div eax, ecx

edx	eax	r/m32(ecx)
0x1	0x2	0x3

Example7.c

```
//Multiply and divide operations  
//when the operand is not a  
//power of two  
//New instructions: imul, div
```

```
int main(){  
    unsigned int a = 1;  
    a = a * 6;  
    a = a / 3;  
    return 0x2bad;  
}
```

	main:	
	push	ebp
	mov	ebp,esp
	push	ecx
	mov	dword ptr [ebp-4],1
	mov	eax,dword ptr [ebp-4]
★	imul	eax,eax,6
	mov	dword ptr [ebp-4],eax
	mov	eax,dword ptr [ebp-4]
	xor	edx,edx
	mov	ecx,3
★	div	eax,ecx
	mov	dword ptr [ebp-4],eax
	mov	eax,2BADh
	mov	esp,ebp
	pop	ebp
	ret	

REP STOS – Repeat Store String

- One of a family of “rep” operations, which repeat a single instruction multiple times. (i.e. “stos” is also a standalone instruction)
 - Rep isn’t technically it’s own instruction, it’s an instruction prefix
- All rep operations use ecx register as a “counter” to determine how many times to loop through the instruction. Each time it executes, it decrements ecx. Once ecx == 0, it continues to the next instruction.
- Either moves one byte at a time or one dword at a time.
- Either fill byte at [edi] with al or fill dword at [edi] with eax.
- Moves the edi register forward one byte or one dword at a time, so that the repeated store operation is storing into consecutive locations.
- So there are 3 pieces which must happen before the actual rep stos occurs: set edi to the start destination, eax/al to the value to store, and ecx to the number of times to store

rep stos Setup

- So what's this going to do?

004113AC lea edi,[ebp-0F0h]
Set edi - the destination

004113B2 mov ecx,3Ch
Set ecx - the count

004113B7 mov eax,0CCCCCCCCCh
Set eax - the value

004113BC rep stos dword ptr es:[edi]
Start the repeated store

REP MOVS – Repeat Move Data String to String

- One of a family of “rep” operations, which repeat a single instruction multiple times. (i.e. “movs” is also a standalone instruction)
- All rep operations use ecx register as a “counter” to determine how many times to loop through the instruction. Each time it executes, it decrements ecx. Once ecx == 0, it continues to the next instruction.
- Either moves one byte at a time or one dword at a time.
- Either move byte at [esi] to byte at [edi] or move dword at [esi] to dword at [edi].
- Moves the esi and edi registers forward one byte or one dword at a time, so that the repeated store operation is storing into consecutive locations.
- So there are 3 pieces which must happen before the actual rep movs occurs: set esi to the start source, set edi to the start destination, and set ecx to the number of times to move

Example9.c

Journey to the center of memcpy()

```
//Journey to the center of memcpy
#include <stdio.h>

typedef struct mystruct{
    int var1;
    char var2[4];
} mystruct_t;

int main(){
    mystruct_t a, b;
    a.var1 = 0xFF;
    memcpy(&b, &a, sizeof(mystruct_t));
    return 0xAce0Ba5e;
}
```

main:

00401010	push	ebp
00401011	mov	ebp,esp
00401013	sub	esp,10h
00401016	mov	dword ptr [a],0FFh
0040101D	push	8
0040101F	lea	eax,[a]
00401022	push	eax
00401023	lea	ecx,[b]
00401026	push	ecx
00401027	call	memcpy (401042h)
0040102C	add	esp,0Ch
0040102F	mov	eax,0ACE0BA5Eh
00401034	mov	esp,ebp
00401036	pop	ebp
00401037	ret	

It begins...

memcpy:

```
push    ebp
mov     ebp,esp
push    edi      ;callee save
push    esi      ;callee save
mov     esi,dword ptr [ebp+0Ch] ;2nd param - source ptr
mov     ecx,dword ptr [ebp+10h] ;3rd param - copy size
mov     edi,dword ptr [ebp+8]   ;1st param - destination ptr
mov     eax,ecx   ;copy length to eax
mov     edx,ecx   ;another copy of length for later use
add     eax,esi   ;eax now points to last byte of src copy
cmp     edi,esi   ;edi (dst) – esi (src) and set flags
jbe    1026ED30 ;jump if ZF = 1 or CF = 1
```

;It will execute different code if the dst == src or if the destination is below (unsigned less than) the source (so jbe is an unsigned edi <= esi check)



1026ED30 cmp ecx,100h ;ecx - 0x100 and set flags

1026ED36 jb 1026ED57 ;jump if CF == 1

;Hmmm...since ecx is the length, it appears to do something different based on whether the length is below 0x100 or not. We could investigate the alternative path later if we wanted.

1026ED57 test edi,3 ;edi AND 0x3 and set flags

1026ED5D jne 1026ED74 ;jump if ZF == 0

;It is checking if either of the lower 2 bits of the destination address are set. That is, if the address ends in 1, 2, or 3. If both bits are 0, then the address can be said to be 4-byte-aligned. so it's going to do something different based on whether the destination is 4-byte-aligned or not.

```
1026ED5F shr      ecx,2 ;divide len by 4
1026ED62 and      edx,3 ;edx still contains a copy of ecx
1026ED65 cmp      ecx,8 ;ecx - 8 and set flags
1026ED68 jb       1026ED94 ;jump if CF == 1
;But we currently don't get to the next instruction 1026ED6A,
instead we jump to 1026ED94... :(
```

★ 1026ED6A rep movs dword ptr es:[edi],dword ptr [esi]
1026ED6C jmp dword ptr [edx*4+1026EE84h]

The rep movs is the target of this expedition.

Q: But how can we reach the rep mov?

A: Need to make it so that (length to copy) / 4 >= 8, so we
don't take the jump below



23 REP MOVS - Repeat Move Data String to String

- One of a family of “rep” operations, which repeat a single instruction multiple times. (i.e. “movs” is also a standalone instruction)
- All rep operations use ecx register as a “counter” to determine how many times to loop through the instruction. Each time it executes, it decrements ecx. Once ecx == 0, it continues to the next instruction.
- Either moves one byte at a time or one dword at a time.
- Either move byte at [esi] to byte at [edi] or move dword at [esi] to dword at [edi].
- Moves the esi and edi registers forward one byte or one dword at a time, so that the repeated store operation is storing into consecutive locations.
- So there are 3 pieces which must happen before the actual rep movs occurs: set esi to the start source, set edi to the start destination, and set ecx to the number of times to move

Some high level pseudo-code approximation

```
memcpy(void * dst, void * src, unsigned int len){  
    if(dst <= src){  
        //Path we didn't take, @ 1026ED28  
    }  
    if(dst & 3 != 0){  
        //Other path we didn't take, @ 1026ED74  
    }  
    if((len / 4) >= 8){  
        ecx = len / 4;  
        rep movs dword dst, src;  
    }  
    else{  
        //sequence of individual mov instructions  
        //as appropriate for the size to be copied  
    }  
    ...  
}
```



LEAVE

- “Set ESP to EBP, then pop EBP”
- Then why haven’t we seen it elsewhere already?
 - Depends on compiler and options

1026EE94	mov	eax,dword ptr [ebp+8]
1026EE97	pop	esi
1026EE98	pop	edi
1026EE99	leave	
1026EE9A	ret	

Instructions we now know(24)

- NOP
- PUSH/POP
- CALL/RET
- MOV/LEA
- ADD/SUB
- JMP/Jcc
- CMP/TEST
- AND/OR/XOR/NOT
- SHR/SHL
- IMUL/DIV
- REP STOS, REP MOV
- LEAVE

Recognizing C Constructs

- Now leaving Xeno's x86 slides and moving into PMA Chapter 6
- Moving from sounding words out to recognizing words and how they form sentences.

Globals vs Locals

```
int x = 1;
int y = 2;

void main()
{
    x = x+y;
    printf("total = %d\n", x);
}
```

Listing 6-1: A simple program with two global variables

```
void main()
{
    int x = 1;
    int y = 2;

    x = x+y;
    printf("total = %d\n", x);
}
```

Listing 6-2: A simple program with two local variables

00401003	mov	eax, dword_40CF60
00401008	add	eax, dword_40C000
0040100E	mov	dword_40CF60 , eax ①
00401013	mov	ecx, dword_40CF60
00401019	push	ecx
0040101A	push	offset aTotalID ;"total = %d\n"
0040101F	call	printf

Listing 6-3: Assembly code for the global variable example in Listing 6-1

Globals vs Locals

```
int x = 1;
int y = 2;

void main()
{
    x = x+y;
    printf("total = %d\n", x);
}
```

Listing 6-1: A simple program with two global variables

```
void main()
{
    int x = 1;
    int y = 2;

    x = x+y;
    printf("total = %d\n", x);
}
```

Listing 6-2: A simple program with two local variables

00401006	mov	dword ptr [ebp-4], 1
0040100D	mov	dword ptr [ebp-8], 2
00401014	mov	eax, [ebp-4]
00401017	add	eax, [ebp-8]
0040101A	mov	[ebp-4], eax
0040101D	mov	ecx, [ebp-4]
00401020	push	ecx
00401021	push	offset aTotalD ; "total = %d\n"
00401026	call	printf

Listing 6-4: Assembly code for the local variable example in Listing 6-2.

Arithmetice Operations

```
int a = 0;
int b = 1;
a = a + 11;
a = a - b;
a--;
b++;
b = a % 3;
```

Listing 6-6: C

00401006	mov	[ebp+var_4], 0
0040100D	mov	[ebp+var_8], 1
00401014	mov	eax, [ebp+var_4] ❶
00401017	add	eax, 0Bh
0040101A	mov	[ebp+var_4], eax
0040101D	mov	ecx, [ebp+var_4]
00401020	sub	ecx, [ebp+var_8] ❷
00401023	mov	[ebp+var_4], ecx
00401026	mov	edx, [ebp+var_4]
00401029	sub	edx, 1 ❸
0040102C	mov	[ebp+var_4], edx
0040102F	mov	eax, [ebp+var_8]
00401032	add	eax, 1 ❹
00401035	mov	[ebp+var_8], eax
00401038	mov	eax, [ebp+var_4]
0040103B	cdq	
0040103C	mov	ecx, 3
00401041	idiv	ecx
00401043	mov	[ebp+var_8], edx ❺

Listing 6-7: Assembly code for the arithmetic exam

If Statements

```
int x = 1;  
int y = 2;  
  
if(x == y){  
    printf("x equals y.\n");  
}else{  
    printf("x is not equal to y.\n");  
}
```

Listing 6-8: C code if statement example

If Statement

00401006	mov	[ebp+var_8], 1
0040100D	mov	[ebp+var_4], 2
00401014	mov	eax, [ebp+var_8]
00401017	cmp	eax, [ebp+var_4] ①
0040101A	jnz	short loc_40102B ②
0040101C	push	offset aXEqualsY_ ; "x equals y.\n"
00401021	call	printf
00401026	add	esp, 4
00401029	jmp	short loc_401038 ③
0040102B loc_40102B:		
0040102B	push	offset aXI IsNotEqualToY ; "x is not equal to y.\n"
00401030	call	printf

```
int x = 1;  
int y = 2;  
  
if(x == y){  
    printf("x equals y.\n");  
}else{  
    printf("x is not equal to y.\n");  
}
```

Listing 6-8: C code if statement example

Listing 6-9: Assembly code for the if statement example in Listing 6-8

Nested If Statements

```
int x = 0;
int y = 1;
int z = 2;

if(x == y){
    if(z==0){
        printf("z is zero and x = y.\n");
    }else{
        printf("z is non-zero and x = y.\n");
    }
}else{
    if(z==0){
        printf("z zero and x != y.\n");
    }else{
        printf("z non-zero and x != y.\n");
    }
}
```

Listing 6-10: C code for a nested if statement

```
int x = 0;
int y = 1;
int z = 2;

if(x == y){
    if(z==0){
        printf("z is zero and x = y.\n");
    }else{
        printf("z is non-zero and x = y.\n");
    }
}else{
    if(z==0){
        printf("z zero and x != y.\n");
    }else{
        printf("z non-zero and x != y.\n");
    }
}
```

Listing 6-10: C code for a nested if statement

00401006	mov	[ebp+var_8], 0
0040100D	mov	[ebp+var_4], 1
00401014	mov	[ebp+var_C], 2
0040101B	mov	eax, [ebp+var_8]
0040101E	cmp	eax, [ebp+var_4]
00401021	jnz	short loc_401047 ①
00401023	cmp	[ebp+var_C], 0
00401027	jnz	short loc_401038 ②
00401029	push	offset aZIsZeroAndXY_ ; "z is zero and x = y.\n"
0040102E	call	printf
00401033	add	esp, 4
00401036	jmp	short loc_401045
00401038 loc_401038:		
00401038	push	offset aZIsNonZeroAndX ; "z is non-zero and x = y.\n"
0040103D	call	printf
00401042	add	esp, 4
00401045 loc_401045:		
00401045	jmp	short loc_401069
00401047 loc_401047:		
00401047	cmp	[ebp+var_C], 0
0040104B	jnz	short loc_40105C ③
0040104D	push	offset aZZeroAndXY_ ; "z zero and x != y.\n"
00401052	call	printf
00401057	add	esp, 4
0040105A	jmp	short loc_401069
0040105C loc_40105C:		
0040105C	push	offset aZNonZeroAndXY_ ; "z non-zero and x != y.\n"
00401061	call	printf00401061

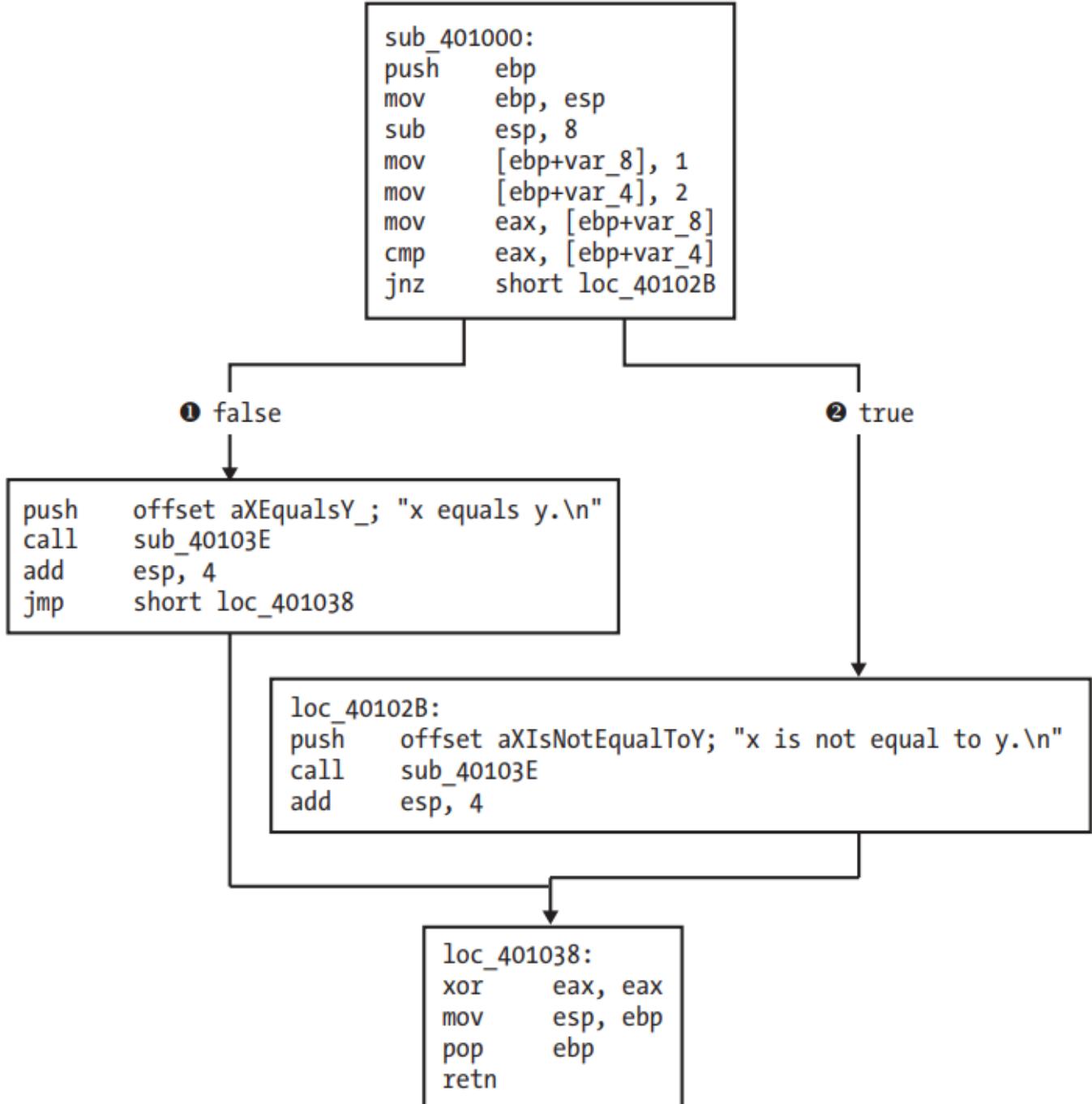
Listing 6-11: Assembly code for the nested if statement example shown in Listing 6-10

```

int x = 0;
int y = 1;
int z = 2;

if(x == y){
    if(z==0){
        printf("z is zero and x = y.\n");
    }else{
        printf("z is non-zero and x = y.\n");
    }
}else{
    if(z==0){
        printf("z zero and x != y.\n");
    }else{
        printf("z non-zero and x != y.\n");
    }
}

```



Listing 6-10: C code for a nested *if* statement

Figure 6-1: Disassembly graph for the *if* statement example in Listing 6-9

For Loop

```
int i;  
  
for(i=0; i<100; i++)  
{  
    printf("i equals %d\n", i);  
}
```

Listing 6-12: C code for a for loop

00401004	mov	[ebp+var_4], 0	①
0040100B	jmp	short loc_401016	②
0040100D	loc_40100D:		
0040100D	mov	eax, [ebp+var_4]	③
00401010	add	eax, 1	
00401013	mov	[ebp+var_4], eax	④
00401016	loc_401016:		
00401016	cmp	[ebp+var_4], 64h	⑤
0040101A	jge	short loc_40102F	⑥
0040101C	mov	ecx, [ebp+var_4]	
0040101F	push	ecx	
00401020	push	offset aID ; "i equals %d\n"	
00401025	call	printf	
0040102A	add	esp, 8	
0040102D	jmp	short loc_40100D	⑦

Listing 6-13: Assembly code for the for loop example in Listing 6-12

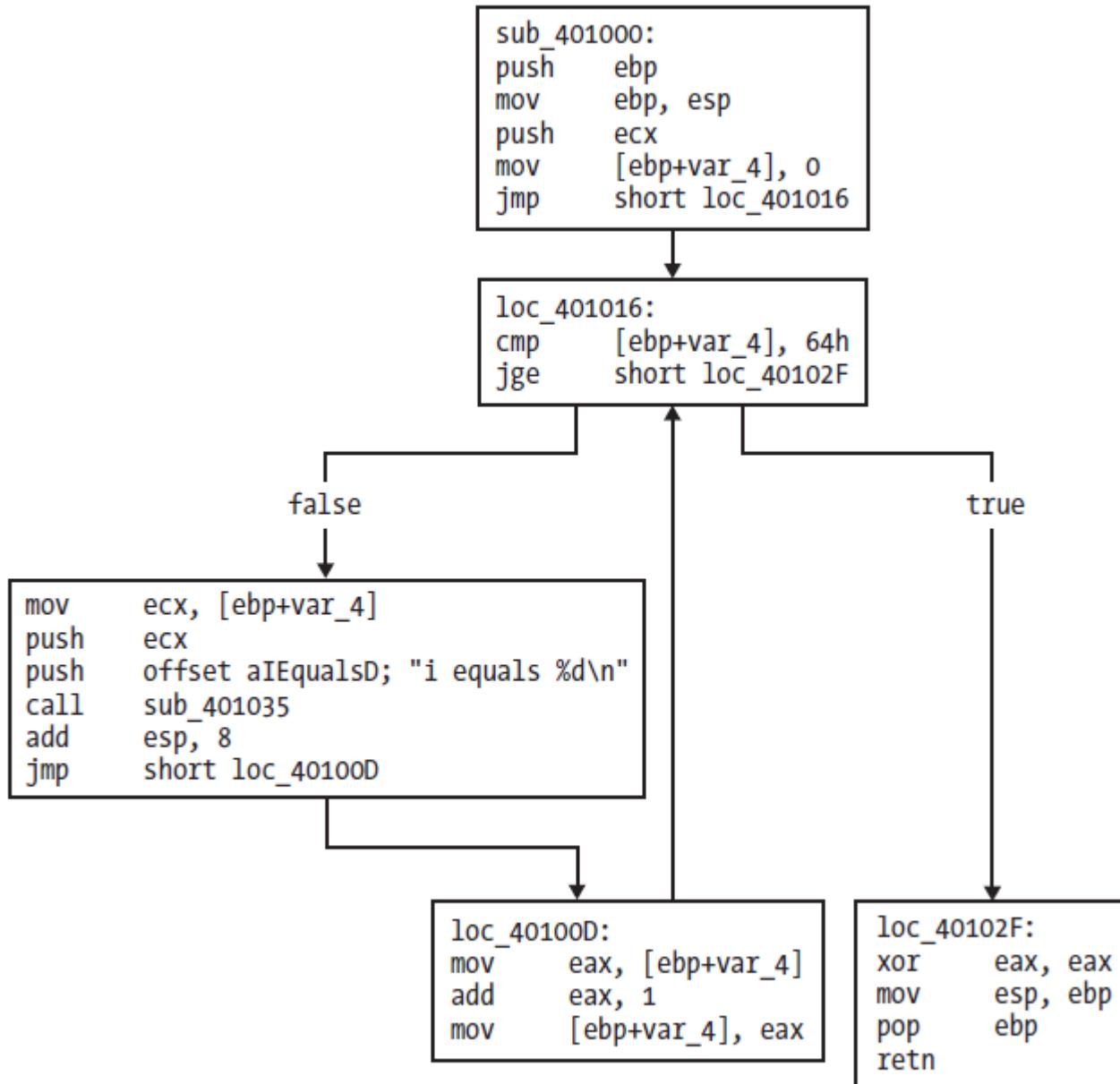


Figure 6-2: Disassembly graph for the for loop example in Listing 6-13

While Loop

```
int status=0;
int result = 0;

while(status == 0){
    result = performAction();
    status = checkResult(result);
}
```

Listing 6-14: C code for a while loop

```
00401036    mov    [ebp+var_4], 0
0040103D    mov    [ebp+var_8], 0
00401044 loc_401044:
00401044    cmp    [ebp+var_4], 0
00401048    jnz    short loc_401063 ①
0040104A    call   performAction
0040104F    mov    [ebp+var_8], eax
00401052    mov    eax, [ebp+var_8]
00401055    push   eax
00401056    call   checkResult
0040105B    add    esp, 4
0040105E    mov    [ebp+var_4], eax
00401061    jmp    short loc_401044 ②
```

Listing 6-15: Assembly code for the while loop example in Listing 6-14

Calling Conventions – An Example

Table 6-1: Assembly Code for a Function Call with Two Different Calling Conventions

Visual Studio version			GCC version		
00401746	mov	[ebp+var_4], 1	00401085	mov	[ebp+var_4], 1
0040174D	mov	[ebp+var_8], 2	0040108C	mov	[ebp+var_8], 2
00401754	mov	eax, [ebp+var_8]	00401093	mov	eax, [ebp+var_8]
00401757	push	eax	00401096	mov	[esp+4], eax
00401758	mov	ecx, [ebp+var_4]	0040109A	mov	eax, [ebp+var_4]
0040175B	push	ecx	0040109D	mov	[esp], eax
0040175C	call	adder	004010A0	call	adder
00401761	add	esp, 8	004010A5	mov	[esp+4], eax
00401764	push	eax	004010A9	mov	[esp], offset TheFunctionRet
00401765	push	offset TheFunctionRet	004010B0	call	printf
0040176A	call	ds:printf			

Switch Statements

```
switch(i)
{
    case 1:
        printf("i = %d", i+1);
        break;
    case 2:
        printf("i = %d", i+2);
        break;
    case 3:
        printf("i = %d", i+3);
        break;
    default:
        break;
}
```

Listing 6-20: C code for a three-option switch statement

00401013	cmp	[ebp+var_8], 1
00401017	jz	short loc_401027 ①
00401019	cmp	[ebp+var_8], 2
0040101D	jz	short loc_40103D
0040101F	cmp	[ebp+var_8], 3
00401023	jz	short loc_401053
00401025	jmp	short loc_401067 ②
00401027 loc_401027:		
00401027	mov	ecx, [ebp+var_4] ③
0040102A	add	ecx, 1
0040102D	push	ecx
0040102E	push	offset unk_40C000 ; i = %d
00401033	call	printf
00401038	add	esp, 8
0040103B	jmp	short loc_401067
0040103D loc_40103D:		
0040103D	mov	edx, [ebp+var_4] ④
00401040	add	edx, 2
00401043	push	edx
00401044	push	offset unk_40C004 ; i = %d
00401049	call	printf
0040104E	add	esp, 8
00401051	jmp	short loc_401067
00401053 loc_401053:		
00401053	mov	eax, [ebp+var_4] ⑤
00401056	add	eax, 3
00401059	push	eax
0040105A	push	offset unk_40C008 ; i = %d
0040105F	call	printf
00401064	add	esp, 8

Listing 6-21: Assembly code for the switch statement example in Listing 6-20

Switch Statement

```
switch(i)
{
    case 1:
        printf("i = %d", i+1);
        break;
    case 2:
        printf("i = %d", i+2);
        break;
    case 3:
        printf("i = %d", i+3);
        break;
    default:
        break;
}
```

Listing 6-20: C code for a three-option switch statement

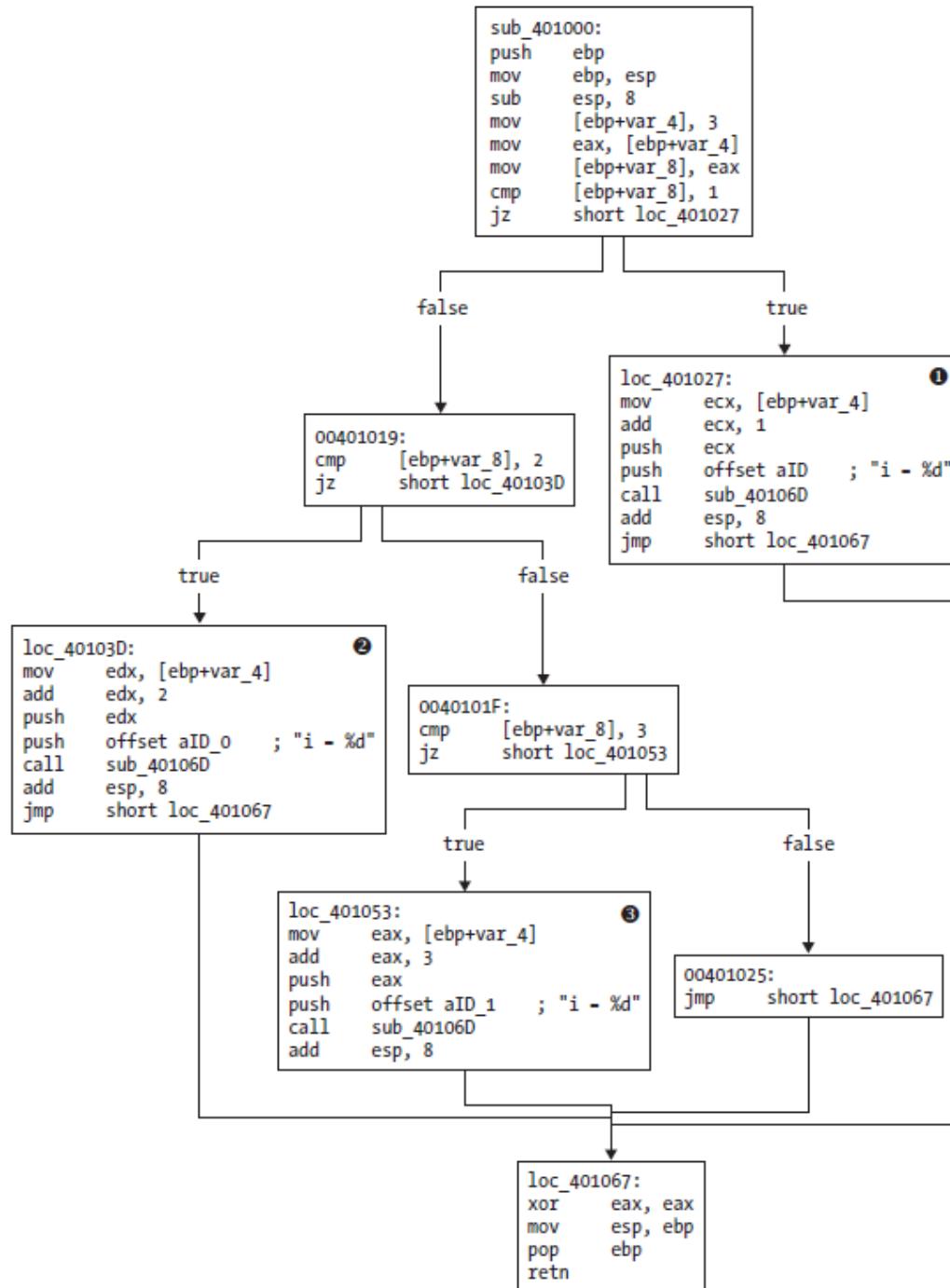


Figure 6-3: Disassembly graph of the if style switch statement example in Listing 6-21

Jump Table

```
switch(i)
{
    case 1:
        printf("i = %d", i+1);
        break;
    case 2:
        printf("i = %d", i+2);
        break;
    case 3:
        printf("i = %d", i+3);
        break;
    case 4:
        printf("i = %d", i+3);
        break;
    default:
        break;
}
```

Listing 6-22: C code for a four-opt

00401016	sub	ecx, 1
00401019	mov	[ebp+var_8], ecx
0040101C	cmp	[ebp+var_8], 3
00401020	ja	short loc_401082
00401022	mov	edx, [ebp+var_8]
00401025	jmp	ds:off_401088[edx*4] ↗
0040102C		loc_40102C:
		...
00401040	jmp	short loc_401082
00401042		loc_401042:
		...
00401056	jmp	short loc_401082
00401058		loc_401058:
		...
0040106C	jmp	short loc_401082
0040106E		loc_40106E:
		...
00401082		loc_401082:
00401082	xor	eax, eax
00401084	mov	esp, ebp
00401086	pop	ebp
00401087		retn
00401087		_main endp
00401088	off_401088 dd offset	loc_40102C
0040108C		dd offset loc_401042
00401090		dd offset loc_401058
00401094		dd offset loc_40106E

Listing 6-23: Assembly code for the switch statement example in Listing 6-22

Jump Table (cont.)

```
switch(i)
{
    case 1:
        printf("i = %d", i+1);
        break;
    case 2:
        printf("i = %d", i+2);
        break;
    case 3:
        printf("i = %d", i+3);
        break;
    case 4:
        printf("i = %d", i+3);
        break;
    default:
        break;
}
```

Listing 6-22: C code for a four-opt

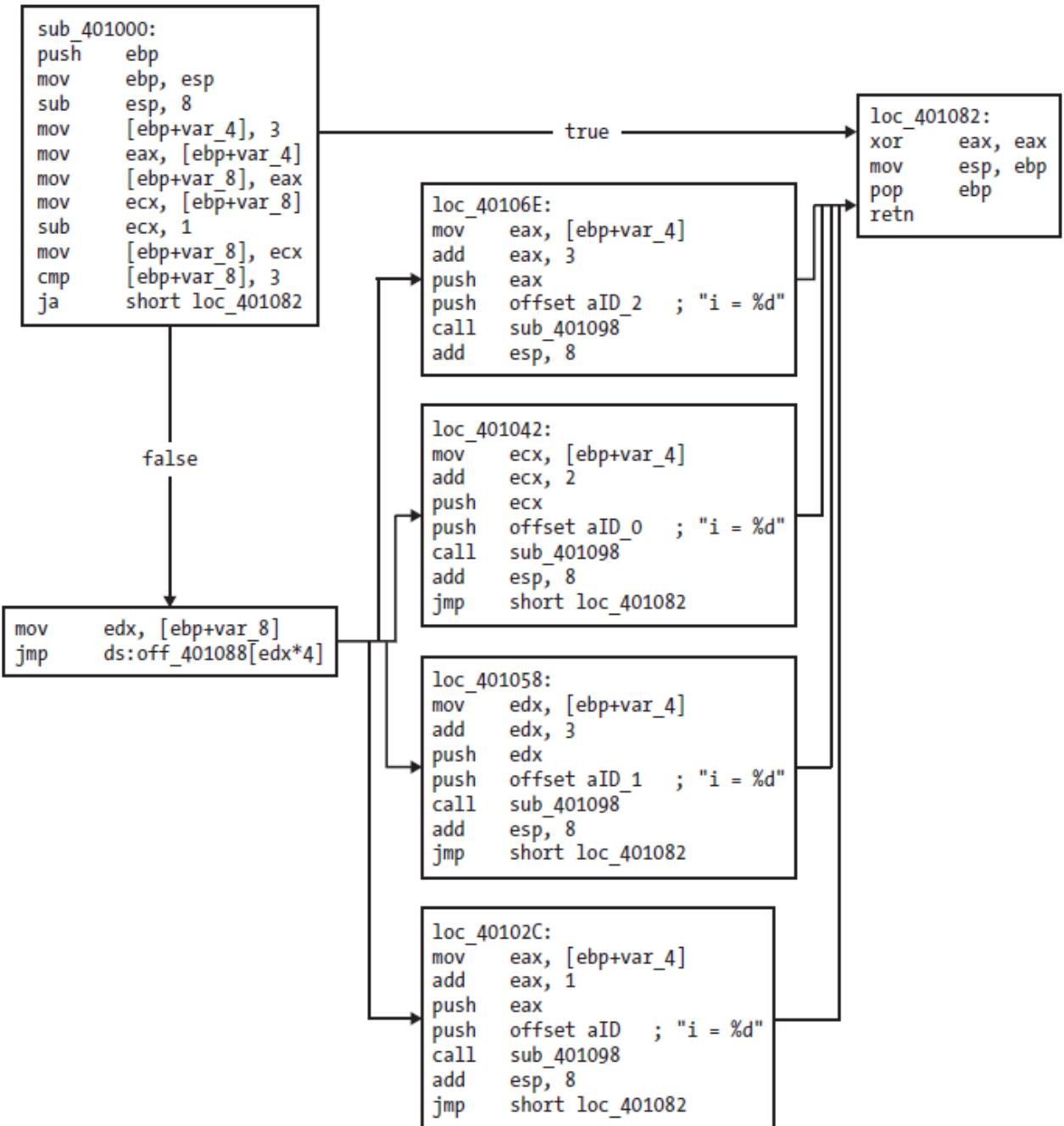


Figure 6-4: Disassembly graph of jump table switch statement example

Arrays

```
int b[5] = {123,87,487,7,978};  
void main()  
{  
    int i;  
    int a[5];  
  
    for(i = 0; i<5; i++)  
    {  
        a[i] = i;  
        b[i] = i;  
    }  
}
```

Listing 6-24: C code for an array

00401006	mov	[ebp+var_18], 0
0040100D	jmp	short loc_401018
0040100F	loc_40100F:	
0040100F	mov	eax, [ebp+var_18]
00401012	add	eax, 1
00401015	mov	[ebp+var_18], eax
00401018	loc_401018:	
00401018	cmp	[ebp+var_18], 5
0040101C	jge	short loc_401037
0040101E	mov	ecx, [ebp+var_18]
00401021	mov	edx, [ebp+var_18]
00401024	mov	[ebp+ecx*4+var_14], edx ①
00401028	mov	eax, [ebp+var_18]
0040102B	mov	ecx, [ebp+var_18]
0040102E	mov	dword_40A000[ecx*4], eax ②
00401035	jmp	short loc_40100F

Listing 6-25: Assembly code for the array in Listing 6-24

Structs

```
struct my_structure { ❶
    int x[5];
    char y;
    double z;
};

struct my_structure *gms; ❷

void test(struct my_structure *q)
{
    int i;
    q->y = 'a';
    q->z = 15.6;
    for(i = 0; i<5; i++){
        q->x[i] = i;
    }
}

void main()
{
    gms = (struct my_structure *) malloc(
        sizeof(struct my_structure));
    test(gms);
}
```

Listing 6-26: C code for a struct example

```
00401050    push    ebp
00401051    mov     ebp, esp
00401053    push    20h
00401055    call    malloc
0040105A    add     esp, 4
0040105D    mov     dword_40EA30, eax
00401062    mov     eax, dword_40EA30
00401067    push    eax ❸
00401068    call    sub_401000
0040106D    add     esp, 4
00401070    xor     eax, eax
00401072    pop     ebp
00401073    retn
```

Listing 6-27: Assembly code for the `main` function

```
00401000    push    ebp
00401001    mov     ebp, esp
00401003    push    ecx
00401004    mov     eax, [ebp+arg_0]
00401007    mov     byte ptr [eax+14h], 61h
0040100B    mov     ecx, [ebp+arg_0]
0040100E    fld     ds:dbl_40B120 ❹
00401014    fstp   qword ptr [ecx+18h]
00401017    mov     [ebp+var_4], 0
0040101E    jmp     short loc_401029
00401020    loc_401020:
00401020    mov     edx, [ebp+var_4]
00401023    add     edx, 1
00401026    mov     [ebp+var_4], edx
00401029    loc_401029:
00401029    cmp     [ebp+var_4], 5
0040102D    jge     short loc_40103D
0040102F    mov     eax, [ebp+var_4]
00401032    mov     ecx, [ebp+arg_0]
00401035    mov     edx, [ebp+var_4]
00401038    mov     [ecx+eax*4], edx ❺
0040103B    jmp     short loc_401020
0040103D    loc_40103D:
0040103D    mov     esp, ebp
0040103F    pop     ebp
00401040    retn
```

Listing 6-28: Assembly code for the `test` function in the struct

Malloc

Syntax

```
void *malloc(  
    size_t size  
>);
```

Parameters

size

Bytes to allocate.

Return Value

`malloc` returns a void pointer to the allocated space, or `NULL` if there is insufficient memory available.

L
I
N
K
E
D

L
I
S
T

```
struct node
{
    int x;
    struct node * next;
};

typedef struct node pnode;

void main()
{
    pnode * curr, * head;
    int i;
    head = NULL;

    for(i=1;i<=10; i++) ①
    {
        curr = (pnode *)malloc(sizeof(pnode));
        curr->x = i;
        curr->next = head;
        head = curr;
    }

    curr = head;

    while(curr) ②
    {
        printf("%d\n", curr->x);
        curr = curr->next ;
    }
}
```

Listing 6-29: C code for a linked list traversal

```
0040106A    mov    [ebp+var_8], 0
00401071    mov    [ebp+var_C], 1
00401078
00401078 loc_401078:
00401078    cmp    [ebp+var_C], 0Ah ←
0040107C    jg     short loc_4010AB
0040107E    mov    [esp+18h+var_18], 8
00401085    call   malloc
0040108A    mov    [ebp+var_4], eax
0040108D    mov    edx, [ebp+var_4]
00401090    mov    eax, [ebp+var_C]
00401093    mov    [edx], eax ①
00401095    mov    edx, [ebp+var_4]
00401098    mov    eax, [ebp+var_8]
0040109B    mov    [edx+4], eax ②
0040109E    mov    eax, [ebp+var_4]
004010A1    mov    [ebp+var_8], eax
004010A4    lea    eax, [ebp+var_C]
004010A7    inc    dword ptr [eax]
004010A9    jmp    short loc_401078

004010AB loc_4010AB:
004010AB    mov    eax, [ebp+var_8]
004010AE    mov    [ebp+var_4], eax
004010B1
004010B1 loc_4010B1:
004010B1    cmp    [ebp+var_4], 0 ③ ←
004010B5    jz     short locret_4010D7
004010B7    mov    eax, [ebp+var_4]
004010BA    mov    eax, [eax]
004010BC    mov    [esp+18h+var_14], eax
004010C0    mov    [esp+18h+var_18], offset aD ; "%d\n"
004010C7    call   printf
004010CC    mov    eax, [ebp+var_4]
004010CF    mov    eax, [eax+4]
004010D2    mov    [ebp+var_4], eax ④
004010D5    jmp    short loc_4010B1 ⑤
```

Listing 6-30: Assembly code for the linked list traversal example in Listing 6-29

Lab Work this Afternoon

- Do the Visual Studio lab first – Do not use Ida
- I've included all of Xeno's slides and examples in the folder "IntroX86" on your Windows 7 VM's desktop – review these if you have time or get stuck.
 - Much more detail on building, compiling, decompiling, and debugging in VS Express in Xeno's part 1 slides
- The PMA Labs will help cement your understanding of assembly and get you started in Ida. You may need to read chapters 5 and 6 to be successful in the labs.
 - The PMA labs should be completed using Ida Demo
- If you have extra time, check out the CMU Bomb Lab, also included in Xeno's folder.

Notes and Resources

- All code snippets or images with the caption “Listing *-*” or “Table *-*” are from Sikorski and Honig’s “Practical Malware Analysis”.
- Slides 5-96 are from Xeno Kovah’s “Introduction to Intel x86 Assembly, Architecture, Applications, and Alliteration”. Slides and video lectures are available on opensecuritytraining.info.
- Questions/Comments/Corrections to Lauren Pearce –
LaurenP@lanl.gov